


For Reference

NOT TO BE TAKEN FROM THIS ROOM

Ex LIBRIS
UNIVERSITATIS
ALBERTAEASIS





Digitized by the Internet Archive
in 2023 with funding from
University of Alberta Library

<https://archive.org/details/Dasgupta1974>

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR : Subrata Dasgupta

TITLE OF THESIS : A High-Level Microprogramming
Language

DEGREE FOR WHICH THESIS WAS PRESENTED : M. Sc.

YEAR THIS DEGREE GRANTED : 1974

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

THE UNIVERSITY OF ALBERTA

A HIGH-LEVEL MICROPROGRAMMING LANGUAGE

by



SUBRATA DASGUPTA

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE
OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

SPRING, 1974

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled "A HIGH-LEVEL MICROPROGRAMMING LANGUAGE" submitted by Subrata Dasgupta in partial fulfillment of the requirements for the degree of Master of Science.

ABSTRACT

This thesis defines a high-level microprogramming language called MICROL. The syntax of MICROL resembles that of ALGOL 60 though it differs considerably in its semantics. The language has been defined formally by means of a Simple Precedence Grammar.

A theoretical framework is established as a basis for the language definition. This framework views a microprogrammable processor as a two-level system. The first level consisting of the control unit constitutes an 'inner computer' which manipulates the components of the second level which are the main memory, operational units and other elements of the data flow. A criterion of 'machine-independence' for microprogramming languages is also established.

The language developed in this thesis extends the previous work of Husson, Eckhouse, Hattori et al, and Chu. The main features of this extension are facilities for defining a processor's timing attributes and data-paths. Furthermore, facilities are provided for microprogramming I/O operations.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Prof. L.W. Jackson whose innumerable suggestions helped in clarifying many of the concepts presented here.

I am also grateful to Prof. J. Tartar and Prof. T.A. Marsland for their valuable comments which contributed substantially in improving the final form of the thesis.

Financial support was generously provided by the Department of Computing Science and the National Research Council of Canada.

Finally, I would like to thank my wife for providing the necessary 'logistical' support during the preparation of the thesis.

TABLE OF CONTENTS

	<u>Page</u>
<u>CHAPTER I : INTRODUCTION</u>	1
1.1 Review	1
1.2 The Necessity of High-Level Micro-programming Languages	2
1.3 Organization of the Thesis	5
<u>CHAPTER II : MICROPROGRAMMING LANGUAGES: A REVIEW OF THE LITERATURE</u>	8
2.1 Introduction	8
2.2 APL	9
2.3 Husson's Language	10
2.4 MPL	11
2.5 MPGS	13
2.6 CDL	15
2.7 Micro-Assembler Languages	16
2.8 Discussion	17
2.9 Conclusions	21
<u>CHAPTER III : THEORETICAL FRAMEWORK</u>	22
3.1 Introduction	22
3.2 The Processor	23
3.3 Timing Attributes of a Processor	28
3.4 Conclusions	32
<u>CHAPTER IV : DECLARATIVE STATEMENTS</u>	35
4.1 Location-Object Declarations	35
4.1.1 Semantic Implications of L-Object Declarations	37

<u>CHAPTER IV</u> (cont'd)	<u>Page</u>
4.2 Files and Status-Objects	40
4.3 Time-Object Declarations	42
4.3.1 Phase Declarations	43
4.3.2 Cycle Declarations	44
4.3.3 Phase-Relations	45
4.3.4 Implications of Time-Object Declarations	49
4.4 Operator Declarations	50
<u>CHAPTER V : EXECUTIONAL STATEMENTS</u>	<u>57</u>
5.1 Preliminary Remarks	57
5.2 Expressions	58
5.3 Statements	67
5.3.1 Assignment Statements	67
5.3.2 Goto Statements	69
5.3.3 Compound Statements and Blocks	70
5.3.4 Conditional Statements	70
5.3.5 Iterative Statements	72
5.3.6 File Transfer Statements	75
5.3.7 File Control Statements	77
5.4 Procedures	81
5.4.1 Procedure Declarations	81
5.4.2 Procedure Statements	82
5.4.3 Semantics of MICROL Procedures	84

	<u>Page</u>
<u>CHAPTER VI : SUMMARY AND RECOMMENDATIONS</u>	86
6.1 A Summary of the Capabilities of MICROL	86
6.2 Further Research	88
REFERENCES	91
APPENDIX A : THE SYNTAX OF MICROL IN SIMPLE PRECEDENCE FORM	96
APPENDIX B : EXAMPLES OF MICROL PROGRAMS	106

LIST OF FIGURES

	<u>Page</u>
Fig. 3-1 : Example of a Location-Object	25
Fig. 3-2 : Schematic Representation of a File	25
Fig. 3-3 : The Basic Machine Cycle	30
Fig. 3-4 : Phase Components of a Basic Machine Cycle	30
Fig. 3-5 : Storage-Cycle Scheme	33
Fig. 3-6 : Phase-Relation Scheme	33

CHAPTER I

INTRODUCTION

1.1 Review

The subject of this thesis is the structure and definition of a proposed high-level microprogramming language called MICROL. The present chapter establishes the foundations for this investigation and reviews briefly the organization of the thesis. Adequate literature reviews and the basic microprogramming concepts have been presented by Husson [16], Wilkes [36], and Rosin [27]. The essence of these reviews is assumed for the purpose of this thesis.

Control memories may have one of several different organizations. At one extreme, each bit of a microinstruction corresponds to a single micro-operation, so that the length of the microinstruction word is of the same order of magnitude as the number of data-paths in the processor. This corresponds to the classical structure as proposed by Wilkes [34,35], and is generally referred to as the method of direct control. At the other extreme, every bit of the microinstruction is utilized to the maximum extent possible, thereby yielding the method of encoded control. Encoding of a microword can be realized by grouping together, mutually exclusive

micro-operations such that each of these groups constitute a field of the microinstruction; this particular organization is called a minimally-encoded one [16,26,32]. In a different form of encoding the meaning of certain fields in the microword is interpreted according to the bit configuration of a separate 'control field': the control field is thus analogous to the op-code in machine language instructions, with different values of the 'op-code' invoking different combinations of micro-operations. This scheme yields a highly-encoded microword organization [26].

1.2 The Necessity of High-Level Microprogramming Languages

The characteristics of different control memory organizations have been studied by several authors [7,16,20,22,26]. These studies indicate that a number of parameters such as the set of machine language instructions being implemented, the nature of the microprogrammed algorithms, and processor and memory technologies, contribute directly to the total efficiency of any given organization. The fact emerges however that nearly all microprocessors facilitating dynamic microprogramming, possess control memories with highly-encoded word organizations [27].

The reason for this bias towards highly-encoded organizations for dynamic microprocessors, may be

attributed to the fact that these organizations possess , the op-code/operand structure of conventional machine language instructions. This permits considerable ease of programming at the user level. In contrast, minimally-encoded or direct-control microwords implicitly recognize the availability of parallel data-paths; thus effective programming of such microinstructions necessitates the utilization of all available parallel operations. For most processors, the problem of coding such 'horizontal' microprograms is understandably complex.

One solution to this problem would be the availability of a high-level microprogramming language. The complexities of timing, asynchronicity, parallel task detection, and various other interactions attendant upon horizontal microprogramming may then be transferred from the domain of the programmer to that of the language compiler. In fact it is reasonable to claim that this would be the only way by which horizontal microprogramming at the user level may be made a practical reality.

In a broader context, the need for a high-level microprogramming language is made evident by exactly the same reason that motivated the development of FORTRAN and other problem-oriented languages: to be able to transform the programming task from a machine oriented activity to one that is problem or algorithm oriented. In particular, recent researches on micro-

programming applications makes the necessity of an algorithmic language even more urgently felt. For example, current investigations include the emulation of environments for general-purpose programming languages [13,24,28], SNOBOL 'machines' [28], support for non-standard arithmetic routines and numerical analysis [28,30], and the emulation of operating system environments [9,33,37]. Such application will be greatly facilitated by the availability of an appropriate language for the expression and specification of microprogrammed algorithms.

Finally, attention is drawn to the pedagogical utility of such languages. At the present time, in spite of numerous efforts to define languages for the description of computer systems, the treatment of computer organizations in the literature remain distinctly expository in nature. Microprogramming 'operates' at precisely that level at which the organization and structure of computer systems are mostly described, i.e. at what Bell and Newell call the 'register-transfer' level [3]. A high-level microprogramming language may thus be regarded as a metalanguage for describing digital machines at this level.

A number of microprogramming languages have been developed in the past few years. These are reviewed in

Chapter II. It will be shown that these languages are only partially adequate, particularly with respect to such aspects as timing characteristics (of processors), I/O, machine-independence, and the data-path restrictions that exist at the microprogram level. A new language, MICROL, is subsequently developed with the objective of permitting a more general set of definitional facilities than are presently available.

1.3 Organization of the Thesis

This thesis is organized into six chapters. Chapter II reviews the literature on microprogramming languages with particular reference to high-level languages. A discussion of some a priori characteristics desirable in such languages is also included.

Chapter III provides the theoretical framework for the present study. This framework establishes the viewpoint that a microprogrammable computer can be regarded as a two-level system, composed of a control component and a processor component. The control component, which contains the control memory and the micro-instruction sequencing logic, is viewed as an inner computer. The processor component contains the main and auxiliary memories, operational units, and other functional elements of the data-flow which are

manipulated by the inner computer. Using this framework, the 'machine-independence' of a microprogramming language is defined by:

Definition 1.1

A microprogramming language L is machine-independent if the syntax and semantics of L are independent of the structure and behaviour of any arbitrary control component.

The structure of MICROL is defined in Chapters IV and V. Finally, in Chapter VI, MICROL is compared with those languages reviewed in Chapter II. The most important features of MICROL are, briefly: an ALGOL-60 type syntax; a set of declarative facilities which permit the definition of a large class of host and target processors irrespective of the control memory organization; executorial statements for I/O microprogramming; and facilities for defining a processor's timing attributes. The latter permits automatic detection of parallelism in microprograms.

The scope of the investigation has been limited to the definition of the language; that is, considerations of implementation and compiler methodology have been excluded. However, in Appendix A, the syntax of MICROL is formally defined using the Backus-Naur notation [23], in the form of a Wirth Weber Simple Precedence

Grammar [1] so that its unambiguity is assured. Appendix B contains some examples of microprograms written in MICROL for the MICRO-1600 and the IBM System/360 Model 30.

CHAPTER II

MICROPROGRAMMING LANGUAGES : A REVIEW OF THE LITERATURE

2.1 Introduction

Most high-level microprogramming languages are similar in that they incorporate standard programming constructs, such as assignment statements, conditional statements and subroutines. The differences between the various microprogramming languages appear to be in the declarative facilities that each provides. These facilities are required for specifying the processor's data flow and timing attributes. In this chapter several microprogramming languages are reviewed. It is shown that these languages are inadequate primarily in their declarative facilities because of the following:

(a) Microprogramming involves the manipulation of data between different classes of memory elements, so that the data-paths interconnecting these memories must be specified.

(b) Microprograms written for direct control or minimally-encoded word organizations necessitate the specification of parallel micro-operations. In software languages for conventional single-processor systems the problem of parallel operations simply does not exist. An earlier study of the conditions for

microprogram parallelism [8] shows the necessity of the processor's timing attributes in determining parallel micro-operations. Thus the timing attributes must be specified in the declarative facilities.

(c) The lack of facilities for microprogrammed I/O operations.

Finally there exists the important question as to how a microprogramming language is to be machine-independent. Since the declarative facilities describe the processor, it is clear that the notion of machine-independence of problem-oriented languages is inadequate for microprogramming languages. Definition 1.1 formally establishes a different criterion for machine-independence.

2.2 APL

One of the earliest algorithmic languages used for describing microprograms is APL developed in 1962 by Iverson [18]. Subsequently, a formal description of the IBM System/360 at the machine language level using APL was published [11].

The main attribute of APL is a concise and powerful notation for the manipulation of arrays which constitute the single composite data-object in the language. Since memories, registers, and other storage elements which form the primary data-structures in microprogramming

may be conveniently represented by arrays, APL appears to be particularly suitable for microprogramming.

In its present form however, it offers two serious disadvantages. The first of these, paradoxically, is its restriction in data-structural capabilities solely to arrays. For reasons shown in Chapter III, this would prove to be inadequate for describing the entire range of objects that need to be represented at the microprogramming level. Secondly, the very compactness and power of the APL operators, making it so effective in certain applications, proves to be detrimental in the writing, debugging, and understanding of microprograms, since the sequencing of the data-path activities that characterize microprograms, are grossly underspecified in APL.

2.3 Husson's Language

A language with a FORTRAN-type syntax, developed by Husson et al [16] is composed basically of a machine description part, and a control program description part. The former contains declarative statements similar to the FORTRAN dimension statement. The control program facility is an adaptation of FORTRAN and PL/1 executable statements, and include the statements ASSIGN, IF, GOTO, DO, CALL, and DECODE. The DECODE statement is

similar to the computed GOTO in FORTRAN. For example, the execution of

```
DECODE (X) L1, L2
```

causes a branch to either 'L1' or 'L2' depending on whether the value of 'X' is 0 or 1 respectively. The ASSIGN statement permits specification and manipulation of three data types, binary, logical, and decimal. The IF, GOTO, DO, and CALL statements are identical to corresponding constructs in FORTRAN and PL/1.

At the time it was reported (1970), Husson's language though not completely defined, signified an important point in the evolution of microprogramming since it was the first explicit formulation of a high-level, procedural, microprogramming language.

Unfortunately the machine description facility is restricted to the representation of memory objects only. Moreover, input-output considerations are excluded from the scope of definition; finally it is not clear as to how timing attributes and parallel operations are characterized in the language.

2.4 MPL

More recently Eckhouse [10] developed MPL which is a block-structured language with a PL/1 type syntax. MPL permits the specification of six types of data

items by means of DECLARE statements, namely:

- (a) machine registers, both real and virtual;
- (b) memories, main and micro;
- (c) local and auxiliary storage;
- (d) 'events' which correspond to testable conditions;
- (e) constants which may be decimal, binary, or hexadecimal numeric constants; and finally,
- (f) variables which may take on constant values.

There are three types of executional statements in MPL:

- (a) the assignment statement whose right-hand side may be arithmetic or Boolean expressions;
- (b) the conditional branch of the form IF.....THEN.....ELSE or IF.....THEN;
- (c) the unconditional GOTO statement.

As in PL/1, the basic building block of MPL is the procedure and the concepts of local and global variables have been preserved in the language, providing its block-structured characteristics. Looping however has to be explicitly programmed.

Like Husson's language, MPL is incompletely defined. For example, there are no explicitly defined constructs for I/O microprogramming, and mechanisms for denoting timing and parallelism are unspecified. In the latter context, it would appear that MPL was designed for the generation of highly-encoded microcode.

2.5 MPGS

Hattori, Yano and Fujino [14] described a language called MPGS which is composed of three parts - the machine description part, the function part, and the microprogram part.

The machine description part allows for the definition of the target machine's hardware, such as registers, subregisters, and memories. It also provides facilities for declaring the correspondence between the target machine and the host machine. This correspondence is described by means of DEFINE statements, whilst the target machine hardware is defined by means of DECLARE statements. For example,

```
DECLARE A(32), B(48);  
  
DEFINE  A = SPM#1-2;  
        B = SPM#3-4;
```

defines two target machine registers, 'A' of 32 bits length, and 'B' of 48 bits length which correspond to the host machine registers 'SPM1', 'SPM2'; and 'SPM3', 'SPM4' respectively.

The microcode generation or translation rule is described in terms of translation control statements and user defined microstatements. The function part of an MPGS program consists of subroutines which are called either from other subroutines or by statements

in the microprogram part. For example, the function part may contain a subroutine named ARITH which when called from elsewhere in the program, executes a set of instructions for performing arithmetic functions, the exact nature of which is determined by the actual parameters in the subroutine call statement.

The statements within the microprogram part of an MPGS program describe the control block of the target machine. In essence, the statements in the microprogram part are calls to subroutines contained in the function part, and may in addition, include the names of the temporary registers and microcommands. As an example, a microprogram named IFETCH may be defined in this part in the following form:

```
IFETCH

    START:

    MEMORY (.....);
    ARITH (.....);
    CONTROL(.....);
    ARITH (.....);
    :
    :
```

where 'MEMORY', 'ARITH', 'CONTROL' correspond to the names of subroutines defined in the function part, and the quantities within the brackets are parameters for these specific subroutine calls.

2.6 CDL

CDL, a computer design language developed by Chu [5], originated primarily as a machine description language for the symbolic expression of logic algorithms. It has been subsequently used for the description of microprograms [6]. The syntax of CDL loosely resembles ALGOL along with certain additional primitives for describing both processor elements and operations. From the viewpoint of microprogramming its most important contribution is its ability to specify explicitly, a processor's timing attributes. This is provided in the following way: the declaration statements include the clock statement whereby a clock cycle and its phase components may be defined. For example,

```
clock P(1 - 3)
```

declares a three phase clock named 'P', the phases being identified as P(1), P(2), and P(3). The relation between a clock phase and one or more microstatements is established by prefixing the statements with a 'label', the latter indicating the phase or phases wherein the statements are activated. For example, a shift operation may be defined by

```
/SHR * P(3)/  A - shr A
```

indicating that the statement on the right, identified

in a CDL program by the name 'SHR', is invoked in the clock phase 'P(3)'.

CDL appears to be one of the few procedural languages in which symbolic representation of timing attributes can be made in the source program. Apart from this however, CDL suffers seriously as a high-level microprogramming language since its use requires knowledge of the microinstruction word organization on the part of the programmer. In this sense it resembles a micro-assembler language.

2.7 Micro-Assembler Languages

The application of micro-assembler languages require knowledge of the control memory organization. Two important examples from this class of languages are, a microprogram simulator ALSIM developed by Young [40], and ANIMIL, developed by Rauscher and Agarwala [25].

ALSIM includes facilities for the definition of the processor elements, and the microinstruction organization in terms of its constituent micro-operations. The executorial facilities provide for the coding of microinstructions in the specified format. Since the organization of microwords may be defined by the programmer, ALSIM is in effect, a general purpose micro-assembler language. Features of ALSIM are further discussed by Sitton [31].

ANIMIL is a 'low-level' language developed specifically for programming horizontal microwords. This is, structurally, a symbolic language designed for a specific processor, the AN/UKY-17 at the US Naval Research Laboratories. Its most interesting feature is that its syntax is defined by an Operator Precedence Grammar [1], and formal treatment of the syntax of languages for horizontal microprogramming is considered in some detail.

2.8 Discussion

The foregoing review raises a fundamental question concerning the machine-independence of such high-level languages. Declarative facilities in both Husson's language and in MPL are primarily concerned with the definition of memory elements and registers which from the 'classical' standpoint of programming languages are inherent attributes of the machine organization. The notion of machine-independence as it is used in the development of problem-oriented languages is clearly violated. A solution to this problem is to define machine-independence of a microprogramming language to be that attribute which permits the language to be used for microprogramming any machine, and be only independent of the structure of the inner computer (see Def.1.1.1).

This necessarily implies that the language constructs must permit the definition of the entire range of objects existing at the microprogram level. These include not only memory elements, but also data-paths, operational units, and timing attributes, since the totality of these objects define a given processor.

It is asserted therefore that neither MPL nor Husson's language are 'machine-independent' in the above sense since they do not provide facilities for defining the total set of components at the micro-programming level. The necessity for such facilities may be demonstrated by the following example:

Consider the following portion of the MPL microprogram [10]:

```

DECLARE (R0,R1,.....) BIT(8)
      ⋮
      MDR BIT (16)
      ⋮
R0 | | R1 = MDR

```

The execution of the assignment statement is only possible if there is a data-path from register 'MDR' to the registers 'R0' and 'R1'. Since the microprogram text does not specify this data-path, the compiler may only know of its existence by having embedded in it, both a description of the registers and a description

of their interconnecting data-paths. It follows that the identifiers 'R0', 'R1', and 'MDR' cannot be arbitrary programmer-defined identifiers, since the compiler cannot determine the correspondence between these identifiers and the register descriptions embedded within it. In other words, for a given machine and a given implementation, the identifiers are fixed.

The same example also illustrates the fact that a microprogramming language whose syntax is similar to the more established programming language upon which it is modelled (FORTRAN in the case of Husson's language; and PL/1 in the case of MPL), may possess quite different semantic characteristics owing to the particular nature of microprogramming. For instance, an assignment statement,

$$A = B$$

is syntactically identical in both PL/1 and MPL. However, the validity of the statement and its correct execution is quite different in the two languages: the statement is executed correctly in PL/1 providing the variables 'A' and 'B' have been previously declared. Execution of the statement in MPL is possible only if there is a data-path between the memory elements represented by the identifiers 'A' and 'B'.

As noted previously, the timing attributes of microprogrammed processors have been excluded from the scope of both MPL and Husson's language. Husson [16] briefly mentions the existence of timing descriptions embedded in the translator, but fails to specify the form of this description or the manner by which it is linked to the source program text during translation. The design and implementation considerations for MPL appear to have excluded the problems of timing and parallelism altogether, so that in effect, the language is restricted in its applications to vertical microprogramming.

At the time of this writing, the above languages represent the two major studies in the area of high-level microprogramming languages. A compiler for MPL has already been written (in SNOBOL 4) for the INTERDATA 3 [10], and a recent paper [28] indicated further implementation studies for the Nanodata QM-1. It is worth noting that both these machines are vertically microprogrammed.

The most important feature of MPGS is the fact that it permits the target environment to be defined in terms of the host environment within the program text. In this sense, it offers greater flexibility than either MPL or Husson's language, since the problem of determining the data paths mentioned earlier, is

obviated by means of DEFINE statements. These statements explicitly specify the data-paths in terms of the host machine. Structurally however, MPGS resembles assembly-type rather than high-level languages. Furthermore, the steps or statements within a microprogram body are defined to correspond to a unit clock cycle of the target machine, but it is not clear as to how the timing attributes of the host machine are reflected in the program structure.

2.9 Conclusions

It was pointed out in section 2.1 that the main differences between the various microprogramming languages are related to the declarative facilities. Summarizing these, Husson's language provides a separate machine description part which defines only memory elements (by means of DIMENSION-type constructs); MPL provides facilities for declaring memories, data-items, testable events, and constants; MPGS permits the specification of the target machine memory elements and their correspondence with the host machine. Chu's CDL appears, in the semantic context, a more complete language than the others since it is possible to define almost all types of functional elements within a processor including timing attributes. None of these languages provide for input-output.

CHAPTER III

THEORETICAL FRAMEWORK

3.1 Introduction

The purpose of this chapter is to establish a framework for the definition of MICROL, using a concept proposed by Glushkov [12] and Ito [17], in which a computer may be partitioned into a control component and a processor component.

The control component functions as an inner computer and contains a 'control unit' and a control memory. The former consists of the logic circuitry that decodes and sequences microinstructions residing in the control memory.

The processor component contains memories (which include only programmable memories, hardware registers, and flip-flops), input-output devices, status-indicators, and operational units. The latter consists of combinational circuits such as adders, decoders, and shifters, as well as simple gates which merely transfer data between different memory elements.

Microprogramming involves the activation of the inner computer and thereby alter the state of the processor. It differs basically from machine language programming in that its environment includes hardware

registers which are not normally referenced at the machine language level. Furthermore, microinstructions in the control memory may be accessed but may not be altered by other microinstructions.

With the viewpoint of the control as an inner computer, it is evident that a machine-independent microprogramming language should be independent of the structure of this inner computer and not of the processor. The latter is in effect, analogous to I/O devices at the level of software programming, and the microprogrammer should be cognizant of the processor in the same manner that the software programmer is cognizant of I/O devices. With this viewpoint, the criterion of machine-independence given by Definition 1.1 is proposed. From this definition it follows that a 'machine-independent' microprogramming language will permit the description and programming of any processor irrespective of the control memory organization, sequencing logic, or timing. The language will thus really be control-independent.

3.2 The Processor

Since the processor must be described by a microprogramming language it is necessary to establish its characteristics. This section discusses the characteristics of processor elements, viz., memories, I/O

devices, status-indicators and operational units.

A bistable device is the smallest addressable memory in a processor. Memories are composed of vectors of bistable devices, and may also be represented as vectors of more 'primitive' memories. In MICROL, bistable devices are represented by bits while more complex memories are termed location-objects. For example, Fig. 3-1 depicts the structure of a main memory.

Input-output devices are referred to as files in MICROL. While they are similar to memories in that they store information, they differ from memories in two respects: firstly, the information unit for an I/O device is a character, in contrast to the binary digit which is the information unit for memories. The smallest addressable element of an I/O device is termed a cell in MICROL, which by definition, may store a single character.

Secondly, a data transfer between an I/O device and a memory involves information flow to or from a single cell of the I/O device, the precise cell location being determined by the position of the 'read-write' head at any given time. The cell whose content is transferred is termed the active cell. Furthermore, because of the physical movement of I/O devices, data transfer to or from an active cell within a sequential I/O device results in a different (normally, the

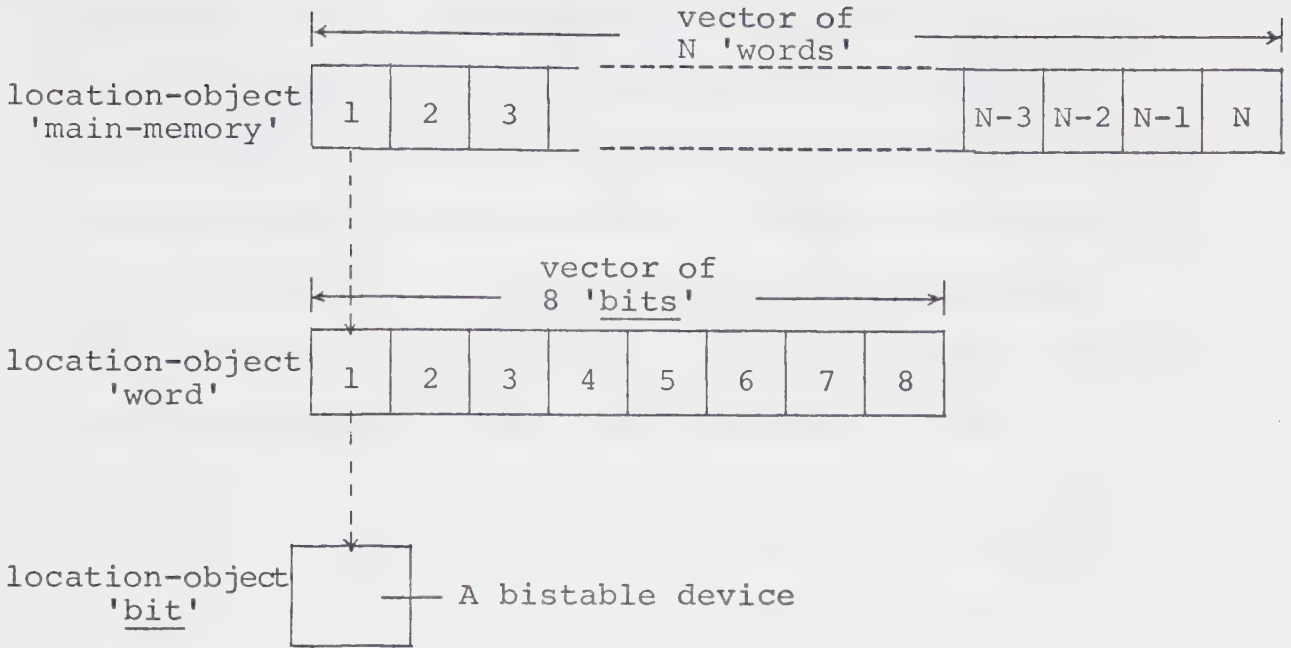


Fig. 3-1

EXAMPLE OF A LOCATION-OBJECT

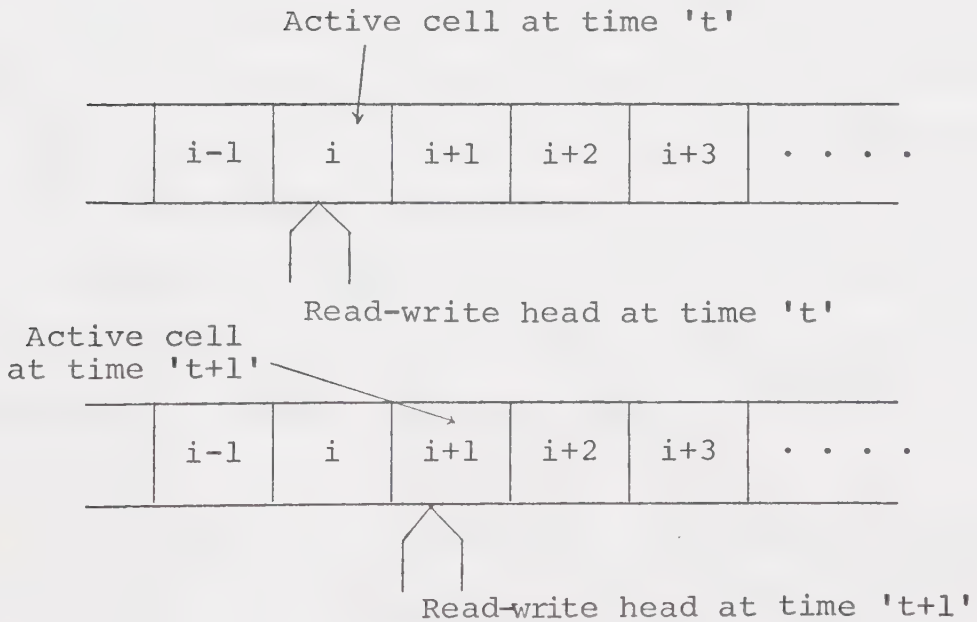


Fig. 3-2

SCHEMATIC REPRESENTATION OF A FILE

adjacent), cell to be activated for the next I/O transfer (Fig. 3-2). For direct access devices, the active cell prior to each data transfer is determined by an address, for example, the track/sector address.

An I/O device may also have a set of status-indicators associated with it. These indicators (termed status-objects in MICROL) store information where the information unit is a binary digit. Status-indicators are 'set' or 'reset' by hardware means, and the microprogrammer may not directly alter their contents. For example, a teletype reader contains a 'keyboard flag' which is set 'on' when a character is made available for transfer. The state of status-indicators may however be tested by microinstructions. Status-indicators may also be present in the processor independent of I/O devices as for example, 'overflow indicators' or 'interrupt flags'. These may also be tested by microinstructions but not accessed otherwise.

Operational units, which are the data-transformational facilities in the processor, are referred to as operators in MICROL. They possess the following properties:

(a) The action of an operational unit may be represented by one or more functions. For example, the addition operation performed by the adder can be represented by the functions:

$$\text{ADD}(i_1, i_2) = (i_1 + i_2)_{\text{mod } 2^n} \quad (3-1)$$

$$\begin{aligned} \text{CARRY}(i_1, i_2) &= 1 \quad \text{if } i_1 + i_2 > 2^n - 1 \\ &= 0 \quad \text{otherwise} \end{aligned} \quad (3-2)$$

where i_1, i_2 are integers. It is assumed in this example that $0 \leq i_1, i_2 \leq 2^n - 1$, and the processor uses two's complement arithmetic.

(b) Operational units are classed as computational, assignment or pointers. A computational unit computes a value in the same class of values as the domain class. Examples of computational units are adders, complementers and shifters.

An assignment unit transfers information from one region of the processor to another. Its action may be represented by the identity function. Examples of assignment units are gates and busses which allow data-flow from one memory to another.

A pointer is a special kind of operational unit in that its function domain is the set of integers, and its function range a set of memory addresses. An example is the combinational logic which determines a core memory location (address) as a function of the integer value contained in the memory address register.

In summary, a computational unit includes the arithmetic, logic, and shifting units, assignment units represent data-paths and busses, and pointers denote

those units in a processor used for accessing random access memories.

(c) The inputs to an operational unit are contained in specific memories, so that these memories are also attributes of the unit. For 'monadic' operational units, there are at least two associated memories ('operands'), one for containing the argument value, the other to store the result. 'Dyadic' operational units require two operands to contain the argument values, and at least one memory for the function value.

The totality of the above properties determine the structure of an operator, and is summarized by the 3-tuple:

$$(F, M_A, M_F) \quad (3-3)$$

where,

F = the set of functions performed by the operational unit;

M_A = the set of memories containing the argument values;

M_F = the set of memories to store the function values.

3.3 Timing Attributes of a Processor

At any given point of time, an operational unit may be 'active' or 'inactive'. In other words, there is a time-interval associated with an operational unit indicating the fraction or phase of some pre-defined

clock cycle within which the unit may be invoked. For instance, recalling the CDL example cited in Chapter II, the state of the operational unit performing a shr is 'active' in phase P(3) of the clock cycle P, and 'inactive' otherwise.

The fundamental time-attribute of a processor is termed the basic machine cycle (abbreviated BMC), which is also referred to as the 'CPU cycle' or 'ALU cycle'. The BMC is representable as a positive number N which is a multiple of some primitive time-unit 'u'. It may also be represented by a sequence of 'N' points on an arbitrary time scale with 0 origin such that the interval between successive points is u (Fig. 3-3).

Adopting this representation, the BMC can be partitioned into n ($n > 0$) successive, not necessarily equal, and not necessarily mutually exclusive segments called phases. For example, a BMC of length N may be partitioned into 3 phases C_1 , C_2 , and C_3 as given in Fig. 3-4. Associated with one or more such phases, are one or more operational units of the processor such that the units are invoked only within these phases.

Thus, a complete definition of the BMC involves a specification of N , which is the value of the BMC as a multiple of some primitive time-unit u , and an ordered set of n pairs of the form (ℓ_i, u_i) , where ℓ_i and u_i are numbers representing the lower and upper bounds

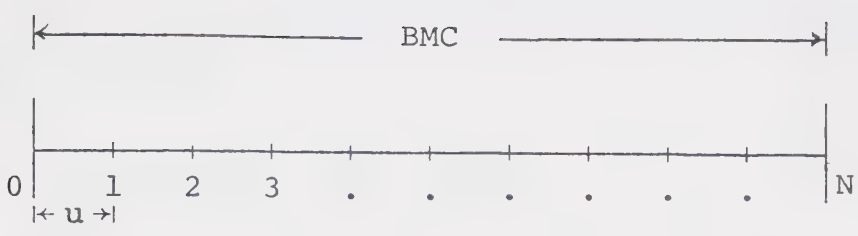


Fig. 3-3

THE BASIC MACHINE CYCLE

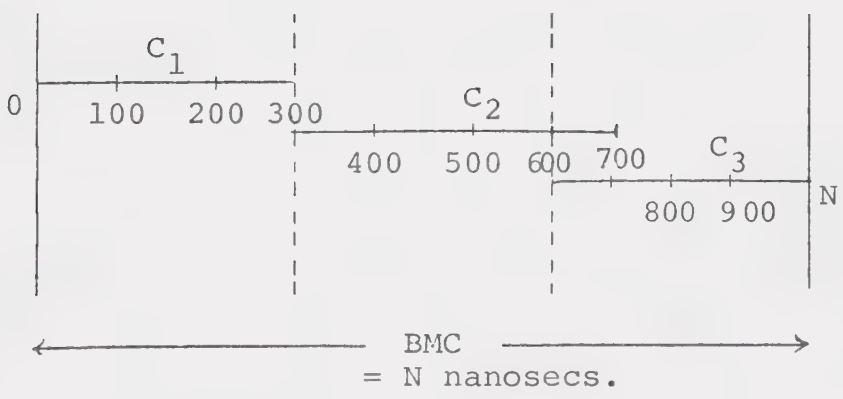


Fig. 3-4

PHASE COMPONENTS OF A BMC

respectively, of the i -th phase of n phases, with $\ell_1 = 0$, and $u_n = N$. For the example of Fig. 3-4, the corresponding quantities are

$$\langle (0, 300), (300, 700), (600, N) \rangle$$

The secondary timing attribute of a processor, is that associated with programmable memories. This timing attribute is termed a storage cycle (abbreviated STC), and there may be a different storage cycle associated with each different programmable memory. Like the BMC, the STC is defined by a value 'N', which is a multiple of some primitive time-unit u' . The STC may be partitioned into two successive, mutually exclusive, but not necessarily equal, segments termed the read-phase and the write-phase respectively, corresponding to the time-validities of storage-read and storage-write operations on the memory. In addition, the read-phase or the write-phase may each be partitioned into smaller phases just as the BMC is partitioned. An example of a storage cycle scheme is that of the IBM system/360 Model 50, schematized in Fig. 3-5 [16]. Here, the read-phase and write-phase are each of 1000 nanosec duration, and the storage cycle value is 2000 nanosec. The read-phase is itself partitioned into two phases P_1 and P_2 .

A tertiary timing attribute of a processor is provided by a set of phase-relations. This arises from

the fact the BMC and the STC's may not be synchronized. For instance, corresponding to each STC there may be four BMC's, such that the start of a storage cycle has a phase difference of K primitive time units with the start of a basic machine cycle. For example, in the System/360 Model 50, the relationships between the BMC and the storage cycles for main memory and local storage are shown in Fig. 3-6.

The main memory STC and the BMC are synchronous; that is, whenever a main memory STC is invoked, it is synchronized with the start of the BMC. As can be seen from Fig. 3-6, each main memory STC corresponds to four BMC's. The storage cycle for the local storage is of the same duration as the BMC. However the local storage cycle is offset from the BMC by 250 nanosec so that any data flow involving the local storage may only be invoked 250 nanosec after the start of a BMC.

It follows from this example, that a complete definition of a processor's timing attribute requires that the relation between the various cycles be also defined. Phase relations are further discussed in Chapter IV.

3.4 Conclusions

In this chapter, a theoretical framework has been presented for the structure of the MICROL language. It has been shown that:

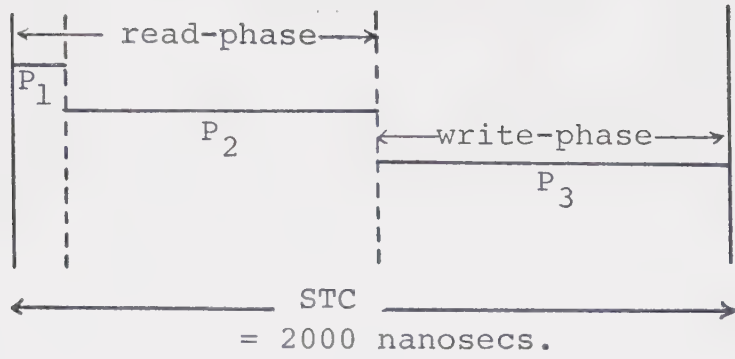


Fig. 3-5

STORAGE CYCLE SCHEME

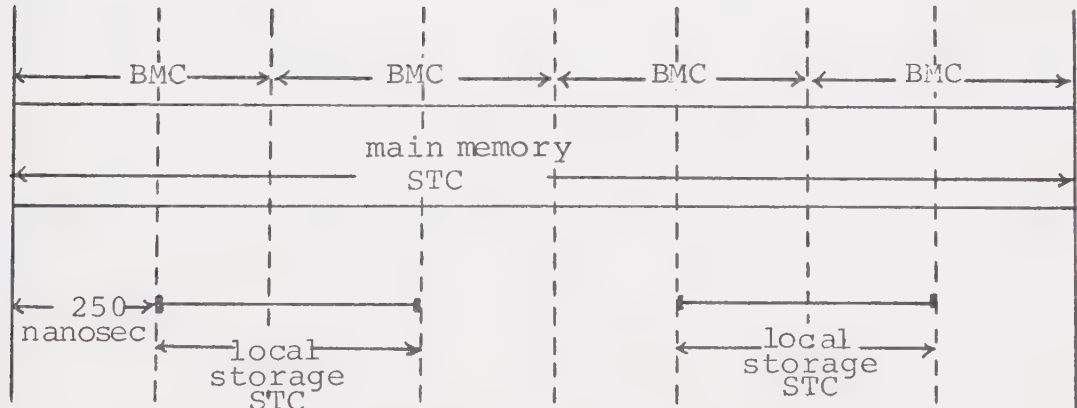


Fig. 3-6

PHASE RELATION SCHEME

- (a) in order to construct a generalized high-level microprogramming language for any processor organization, the property of machine-independence for the language reduces to that of control-independence; and
- (b) by partitioning the computer into a processor and a control, the components that have to be defined are memories, I/O devices, status-indicators, operational units, and time attributes. The next two chapters apply the results of this analysis to the definition of MICROL.

CHAPTER IV

DECLARATIVE STATEMENTS

4.1 Location-Object Declarations

The MICROL programmer uses location-objects (or ℓ -objects) in referring to addressable memories. All ℓ -objects are formed from the bit which represents any bistable device. Location-object declarations are of the form

$$\underline{\text{loc}}\ L = T \qquad (4-1)$$

where 'loc L=' is the ℓ -object symbol clause, and 'T' is the ℓ -object type clause, which may be simple or complex. The symbol 'L' is any programmer-defined identifier designating the name of the declared ℓ -object.

If 'T' is simple, then it specifies either the name of a previously defined ℓ -object or a bit, and the declaration is termed a simple ℓ -object declaration. When 'T' is a bit, the declaration identifies an ℓ -object named 'L' possessing the data-structure of a bit. If 'T' is the name of a previously declared ℓ -object, the declaration creates a synonym for the ℓ -object named 'T'.

If 'T' is complex, the declaration is termed a composite ℓ -object declaration, and defines new ℓ -objects according to the following composition rules:

(a) Sequences

If ' L_1 ' is a primitive bit, or a previously declared ℓ -object, a composite ℓ -object ' L ' may be defined by means of the sequence constructor:

$$\underline{\text{loc}} L = \underline{\text{seq}} (i) L_1 \quad (4-2)$$

where ' i ' is a positive integer specifying the length of the sequence.

The ℓ -object ' L ' in eqn. (4-2) is defined as a vector of i elements, each element being identical in structure to ' L_1 '. An element of this sequence is addressed by subscripting the name ' L ' by an integer ' n ', where $0 < n \leq i$.

(b) Tuples

If ' L_1 ', ' L_2 ', ..., ' L_n ' ($n \geq 1$), are names of ℓ -objects, then a composite ℓ -object named ' L ' may be created by means of the tuple constructor:

$$\underline{\text{loc}} L = (L_1, L_2, \dots, L_n) \quad (4-3)$$

Each element ' L_i ' of such an n -tuple is a field. This declaration creates a data-structure that is effectively a concatenation of the ℓ -objects named in the tuple list. Any subsequent reference to ' L ' addresses the entire set of ℓ -objects ' L_1 ', ' L_2 ', ..., ' L_n ' as a single concatenated object.

(c) Union

A new ℓ -object may be defined as the union of a set of previously defined ℓ -objects according to the syntax

$$\underline{\text{loc}} L = \underline{\text{union}} (L_1, L_2, \dots, L_n) \quad (4-4)$$

Semantically, an ℓ -object declared as an union is identical to the logical union of sets,

$$L_1 \cup L_2 \cup \dots \cup L_n$$

such that a subsequent reference to 'L' refers to all the names in the union list.

A set of ℓ -objects may be declared in the form of a multiple declaration according to the following syntactic form:

$$\underline{\text{loc}} L_1 = \underline{\text{loc}} L_2 = \dots = \underline{\text{loc}} L_n = T \quad (4-5)$$

where ' L_1 ', ' L_2 ', \dots ', ' L_n ' are names of the newly declared ℓ -objects, and 'T' is the common type clause, which may be in any of the forms described above. It follows that the ℓ -objects defined by eqn. (4-5) possess identical data structures.

4.1.1 Semantic Implications of L-Object Declarations

The declaration of ℓ -objects in MICROL is somewhat analogous to the declaration of variables in

programming languages. There are however, important semantic differences. From the viewpoint of machine organizations, the role of the declaration is to identify certain memory elements or combinations of memory elements. The declaration:

- (a) defines a data structure; and
- (b) creates external names that identify the data-structures such that the names may be used as variables by the executorial statements in MICROL so as to manipulate these data-structures.

For example, in the ℓ -object declarations:

loc WORD = seq (8) bit

loc MEMORY = seq (512) WORD

the first statement defines an object named 'WORD' consisting of a vector of 8 primitive bits, and the second statement declares an ℓ -object 'MEMORY' consisting of a vector of 512 ℓ -objects having the same structure as a 'WORD'. It is important to note that each ℓ -object declared as a sequence defines a new ℓ -object. For example, the multiple declaration,

loc ADDREG = loc ACC = seq (16) bit

creates two different objects named 'ADDREG' and 'ACC' respectively, though their data structures are identical.

As an example of an ℓ -object defined by means of the tuple composition rule, consider two registers 'AHIGH' and 'ALOW' which serve to store the eight high- and eight low-order bits of the main memory address. Then the main memory address register may be defined in MICROL by:

```
loc AHIGH = loc ALOW = seq (8) bit

loc MAR   = (AHIGH, ALOW)
```

An ℓ -object defined by the union rule is given by the following example:

```
loc GPR = union (R1, R2, R3)
```

This declaration defines an ℓ -object named 'GPR' as the logical union of the ℓ -objects 'R1', 'R2' and 'R3'.

Suppose 'R1', 'R2' and 'R3' are registers each of which provide the 'left' input to the arithmetic and logic units. In declaring the operators (see section 4.4), the MICROL programmer may use the single name 'GPR' in specifying the 'left' operand to the operators instead of the individual names 'R1', 'R2', and 'R3'. By doing so, it is signified that the 'left' operand to the operators is 'R1' or 'R2' or 'R3'.

It was stated earlier that synonyms for previously declared ℓ -objects may be defined using the simple

ℓ-object declaration. As an example the following declarations are considered:

```
loc REG  = seq (8) bit
loc FILE = seq (16) REG
loc CTR  = FILE [16]
```

The first statement declares an ℓ-object named 'REG' as a vector of 8 bits. The next declaration defines a new ℓ-object named 'FILE' as a vector of 16 ℓ-objects each of which has the same structure as 'REG'. The third statement is a simple declaration, which associates the name 'CTR' with the sixteenth element of the ℓ-object 'FILE'. Any subsequent reference to this element can be made using either 'CTR' or 'FILE [16]'.

4.2 Files and Status Objects

In MICROL, files are used in referring to I/O devices. A file is identified by a file name which is of the form

file-id i (4-6)

where 'i' is an integer, and 'file-id' is one of the following MICROL defined names:

- (a) cardr - denotes a card reader;
- (b) cardp - denotes a card punch;

- (c) mtape - denotes a magnetic tape;
- (d) disk - denotes a disk;
- (e) drum - denotes a drum;
- (f) line - denotes a line printer;
- (g) ttr - denotes a teletype reader;
- (h) ttp - denotes a teletype printer;
- (i) ptaper - denotes a paper tape reader;
- (j) ptapep - denotes a paper tape punch.

Unlike ℓ -objects, file names, or more precisely, the integer 'i' in eqn. (4-6) may not be arbitrarily defined by the programmer. To see why, consider an I/O device. The device has a set of status-indicators associated with it, as for example, a 'device-busy flag'. These indicators are set or reset by the hardware.

If the file names are programmer-defined, it follows that (a) the names of the associated status-indicators would have to be defined by the programmer; and (b) since specific functions are associated with specific status-indicators, the function or meaning of these indicators would also have to be programmer-defined.

It is clearly not possible to define the meaning of status-indicators in the language, hence the name and meaning of status-indicators is specified by the implementation. Consequently, the name of the specific file associated with pre-defined status-indicator names

has also to be implementation-defined. A further complication may arise when two files of the same type possess quite different operating characteristics, for instance, a moveable-head disk, and a fixed-head disk. By defining file names during implementation, specific file names may be associated with specific operating characteristics.

In MICROL, status-indicators are referred to as status-objects (or s-objects). The syntax of MICROL file declarations are given by:

$$\underline{\text{file}} \text{ F } \underline{\text{with}} (s_1, s_2, \dots, s_k) \quad (4-7)$$

where 'F' is the file-name, and 's₁', 's₂', ..., 's_k' are names of s-objects. For example,

file TTR01 with (KBF)

defines a teletype reader.

4.3 Time-Object Declarations

The timing attributes of a processor were discussed in section 3.3. In MICROL, time-objects are used to specify these attributes. The language-defined primitive time-unit is the nanosec, i.e. 10^{-9} seconds. Other time-units that are integer multiples may be constructed by the programmer by means of a time-unit declaration of the form:

timeunit u = i nanosec

where 'u' is a programmer-defined identifier denoting the name of the new time-unit, and 'i' is a positive integer.

4.3.1 Phase Declarations

Both the basic machine cycle and the storage cycle may be partitioned into segments called phases. A phase declaration, possessing the following syntax:

$$\text{phase } P = (t^{\ell} : t^u) u \quad (4-9)$$

where

- 'P' is the programmer-defined name of the phase;
- 'u' is a previously defined time-unit, or the primitive time-unit, nanosec;
- ' t^{ℓ} ' denotes the lower-bound of the phase, in terms of the number of time-units u, after which 'P' starts within a cycle;
- ' t^u ' denotes the upper-bound of the phase, in terms of the number of time-units u, after which 'P' terminates within a cycle.

Read- and write-phases are two special kinds of phases, which are defined according to the syntax:

$$\text{rphase } R = (P_1, P_2, \dots, P_s) \quad (4-10)$$

$$\text{wphase } W = (Q_1, Q_2, \dots, Q_t) \quad (4-11)$$

where 'R' and 'W' are programmer-defined names of the read- and write-phase respectively; ' P_1 ', ..., ' P_s ' are previously declared phases which are used to define 'R'; and ' Q_1 ', ..., ' Q_t ' are previously declared phases which are used to define 'W'.

As examples of phase declarations, the timing scheme shown in Fig. 3-5 can be represented by the following declarations:

```

phase   $P_1$  = (0 : 50) nanosec
phase   $P_2$  = (50 : 1000) nanosec
phase  $P_3$   = (1000 : 2000) nanosec
rphase  $R_1$  = ( $P_1, P_2$ )
wphase  $W_1$  = ( $P_3$ )

```

4.3.2 Cycle Declarations

The basic machine cycle is defined by means of the bmc declaration:

$$\underline{\text{bmc}} \ B = (P_1, P_2, \dots, P_q) \quad (4-12)$$

where 'B' is a programmer-defined name for the BMC, and ' P_1 ', ' P_2 ', ..., ' P_q ' denote previously declared phases. A declaration of the form (4-12) associates a name 'B' with the processor's basic machine cycle. Furthermore it defines the composition of the BMC in terms of predefined phases. As a specific example, the

following declarations completely define the basic machine cycle shown in Fig. 3-4, where it is assumed that the time-unit is nanosec:

$$\begin{aligned}\underline{\text{phase}} \ C_1 &= (0 : 300) \ \underline{\text{nanosec}} \\ \underline{\text{phase}} \ C_2 &= (300 : 700) \ \underline{\text{nanosec}} \\ \underline{\text{phase}} \ C_3 &= (600 : 1000) \ \underline{\text{nanosec}} \\ \underline{\text{bmc}} \ B &= (C_1, C_2, C_3)\end{aligned}$$

Similarly, a storage cycle is defined by means of the declaration

$$\underline{\text{stc}} \ S = (P_1, P_2, \dots, P_k) \quad (4-13)$$

where 'S' is the newly defined name for the storage cycle, and 'P₁', 'P₂', ..., 'P_k' are phase names. A special form of the STC declaration is given by

$$\underline{\text{stc}} \ S = (R, W) \quad (4-14)$$

where 'R' and 'W' are read- and write-phases respectively.

4.3.3 Phase-Relations

The notion of phase-relations was introduced in section 3.3. Phase-relations identify the lag between the start of a basic machine cycle, or the individual phases of the BMC, and the start of a storage cycle or

its phase components. A phase-relation declaration is of one of the following forms:

$$\underline{\text{sync}} P_1 \underline{\text{with}} P_2 \quad (4-15)$$

$$\underline{\text{sync}} P_1 \underline{\text{with}} t^k \quad (4-16)$$

where ' P_1 ' is the name of a storage cycle or a phase of the storage cycle, ' P_2 ' is the name of the BMC or a component phase, and ' t^k ', an integer, is a specific point in the BMC. The semantics of a phase-relation declaration is a function of the precise type of time-objects contained in the declaration, and is given as follows:

(a) The declaration may be of the form (4-15), where ' P_1 ' is the name of a storage cycle, and ' P_2 ', the name of the basic machine cycle. Such a declaration implies that the activation of the entire STC is wholly synchronous with the activation of the BMC. This may be realized in two ways: (i) either the lengths of ' P_1 ' and ' P_2 ' are equal, or (ii) the lengths are unequal but whenever an STC is requested (for example by means of a memory-read operation) it will be invoked at the start of the first phase of the BMC. It follows that there may be any number of BMC's between the invocation of two consecutive STC's.

(b) The phase-relation may be defined between a storage cycle ' P_1 ', and a phase component ' P_2 ' of the

BMC. The activation of ' P_1 ' is synchronized with the invocation of a specific phase ' P_2 ' of the BMC, so that an STC whenever requested, will always lag the BMC by an amount corresponding to the lower bound of ' P_2 '.

(c) The declaration may be of the form (4-16) where ' P_1 ' is a storage cycle. This is similar to the phase-relation given in (b) except that the STC is synchronous with a specific point, t^k , of the BMC, and thus lags the latter by the amount ' t^k '.

It is important to note that the phase-relation semantics indicated in (a), (b) and (c) above, imply that the storage cycle once invoked, runs to completion. That is, if the STC has been declared according to (4-14), the read-phase is immediately followed by the write-phase; similarly, if the STC is defined according to (4-13), the component phases ' P_1 ', ' P_2 ', ..., ' P_k ' follow each other in immediate succession. This is in contrast to the semantics of the remaining phase-relations described below.

(d) If 'R' and 'W' are previously defined read- and write-phases respectively, phase-relations may be declared by:

sync R with P_2

sync W with P_2

where ' P_2 ' is the BMC. These indicate that the read- and write-phases are each synchronized with the BMC such that they are independent of each other: there may be any number of basic machine cycles between the read- and write-phases. This relationship reflects the condition for the 'split-cycle' principle [15].

(e) Phase-relation may be defined between the read- and write-phases of the STC, and the phase components of the BMC:

sync R with P_1

sync W with P_2

where ' P_1 ' and ' P_2 ' are phases of the BMC. Semantically, phase-relations declared in this form imply that, since the write-phase necessarily succeeds the read-phase within the total storage cycle, then if ' P_1 ' and ' P_2 ' are such that ' P_2 ' precedes ' P_1 ' in the BMC, the write-phase 'W' will be synchronous with the phase ' P_2 ' of the succeeding BMC.

(f) Similarly, phase relations may be defined between the read- and write-phases on one hand and specific timing points of the BMC (eqn. 4-16). The semantics of such declarations are identical to those discussed in (e).

(g) Finally, for storage cycles containing a number of phases (eqn. 4-13), phase-relations may be defined

between the phases of the STC and the BMC or its component phases. A phase relation declaration for a specific STC phase is valid for that phase only. Thus, for an STC consisting of k phases, there will be k distinct declarations, one for each phase.

As examples of phase relations, the timing characteristics shown in Fig. 3-6 are referred to. If 'B', 'M' and 'L' are pre-defined names for the BMC, the main memory STC, and the local store STC respectively, then the following phase relations may be declared:

sync M with B

sync L with 250 .

4.3.4 Implications of Time-Object Declarations

Unlike ℓ -objects which correspond to variables, time-objects have no analogous entities in problem-oriented languages. Each quantity defined by means of a time-object declaration, has its correspondence with a distinct timing-attribute of the 'host' processor - the processor being microprogrammed. The execution of a set of microprograms results in the emulation of some 'target' machine; that is, this process activates the host processor so that it behaves like the target machine where 'behaviour' in this context refers only

to the computations that the target machine is capable of performing. The timing attributes of the target machine are not reflected in the emulation process.

By means of time-object declarations, the time-validities of operators may be defined (see section 4.4). These time-validities are necessary for generating microcode, or more specifically, for the detection of parallel micro-operations [8]. This is further discussed in Chapter VI. Time-objects are thus required for translating a MICROL program into object microcode.

4.4 Operator Declarations

Operators are declared in MICROL according to the syntactic form

$$\underline{\text{op}} \ O_L \ \underline{\text{width}} \ W \ \underline{\text{bet}} \ T_\ell \ \underline{\text{and}} \ T_u \ \underline{\text{from}} \ L_I \ \underline{\text{to}} \ L_O \quad (4-17)$$

where 'op O_L ' is the operator symbol clause, 'width W ' is the width clause, 'bet T_ℓ and T_u ' is the operator time clause, and 'from L_I to L_O ' represents the operator parameter clause.

In the above, ' O_L ' represents a list of operators of the form

$$O_1, O_2, \dots, O_m \quad (4-18)$$

such that each O_i is a language-defined operator, and all members of the list possess the common attributes characterized by the width, time, and parameter clauses. MICROL provides an operator vocabulary as follows:

- (i) The arithmetic operators '+', 'plus', 'add', '-', 'minus', 'subtr', whose domain and range are integers. The first three operators denote the ADD function, and the last three, the SUBTRACT function. For any specific implementation of MICROL, the precise semantics of the symbols will be implementation-defined, depending on the type of arithmetic that the processor uses.
- (ii) The Boolean operators '^' (and), 'v' (or), '⊕' (exclusive-or), and '¬' (not), whose domain and range are binary numbers (i.e. bit strings). The '¬' operator is monadic.
- (iii) The shift operators 'shl' (shift-left), 'shr' (shift-right), 'shli' (shift-left-and-insert), 'shri' (shift-right-and-insert), 'shlc' (shift-left-circular) and 'shrc' (shift-right-circular). These operators are also defined over bit strings, and they are all monadic operators. The semantics of these operator symbols may also be defined by the implementation. For instance, in a particular implementation, the operators may invoke the following actions: 'shl' and 'shr' causes the shifted-off bit to be retained in

an overflow register (if the latter is so declared in the parameter clause), or is otherwise lost; shri and shli invoke the same actions except that the current content of the overflow register is inserted in the high-order and low-order bit positions respectively; finally, in the case of shlc and shrc, the shifted-off bit is inserted in the low-order and high-order bit positions respectively.

(iv) The assignment operator ':=', whose action is to transfer the contents of one ℓ -object to another; and

(v) The pointer operator 'ptr', which maps from a domain of integers into a set of addresses as described in section 3.2.

In the width clause, 'W' is an integer. For arithmetic operators, 'W' denotes, as a power of 2, the integer domain. For Boolean, shift, and assignment operators, it indicates the length of the bit strings that these operators act upon. For the ptr operator, the width clause is specified as 'width 0' since this attribute has no significance for this operator.

The time clause defines the time-validity for the operators in O_L ; ' T_ℓ ' and ' T_u ' identify the lower and upper time-bounds for the specified operators, and refer to phases, the basic machine cycle, or storage cycles. For pointer operators, the time clause may be

specified as 'from 0 to 0' if the pointer is invoked by means of some other operation, for example, memory-read or memory-write.

In the parameter clause, ' L_I ' is a string of the form

$$(L_1^1, L_2^1, \dots, L_k^1) : (L_1^2, L_2^2, \dots, L_p^2) \quad (4-19)$$

or

$$(L_1, L_2, \dots, L_q) \quad (4-20)$$

where (4-19) holds for dyadic operators, and (4-20) is for monadic operators. The string (4-19) defines the set of ℓ -objects that are inputs to the two arguments of the operators given in O_L - for example, given the ℓ -objects 'REGA', 'REGB' and 'REGC', and the operators '+' and '-', then the declaration

op +, - width 8 bet T_1 and T_2 from (REGA,REGB);(REGC) to...

indicate that the arithmetic unit has as its 'left' input, 'REGA' or 'REGB', and as its 'right' input, 'REGC'. Since both '+' and '-' are dyadic operators two sets of arguments have to be specified for them.

Similarly, for a monadic operator, there is only one argument, and the string (4-20) will thus be specified as a single set of ℓ -objects.

The quantity ' L_O ' denotes the set of ℓ -objects that may store the function value resulting from the operation:

$$(L_1^3, L_2^3, \dots, L_r^3) : (L_1^4, L_2^4, \dots, L_s^4) \quad (4-21)$$

or

$$(L_1, L_2, \dots, L_t)$$

Eqn. (4-21) is used for operators whose functions include the 'carry' or 'overflow' function as in the instance of arithmetic operators (see eqn. 3-2), or shift-operators. The set ' $L_1^4, L_2^4, \dots, L_s^4$ ' are the ℓ -objects which serve to store the overflow, whilst ' $L_1^3, L_2^3, \dots, L_r^3$ ' contain the result of the operator's primary function. Eqn. (4-22) is used for those operators having no 'carry' function.

Operators are MICROL representation of combinational circuits or operational units in the processor. For purposes of this representation, the operational units may be viewed as 'black boxes' with one or two inputs, and one or two outputs. The strings (4-19) or (4-20) in the declaration specify the input ℓ -objects, whilst (4-21) or (4-22) define the output ℓ -objects, L_i^3 specifying the primary ℓ -objects, and L_j^4 identifying the ℓ -objects for the overflow value. When output ℓ -objects are defined according to (4-22), there are no overflows.

It will be noted that a 'bus' in a processor may be designated in MICROL either as an ℓ -object or as an assignment operator ':='.

Like time-objects, operators characterize the host processor, and serve to identify the valid data-paths between ℓ -objects (by means of the assignment operator), valid functional capabilities of the processor (as a result of defining the computational operators), and the relation between specific ℓ -objects and their address registers (by means of pointers).

It is also to be noted that in any particular implementation, additional operators to those described here may be defined and added to the operator vocabulary, providing that they may be included within the classes defined. For example, if in a particular processor, there exists a hardware multiplier, then an operator '*' may be added to the set of arithmetic operators, and subsequently, may be declared by the MICROL programmer, provided the semantics are implementation-defined.

The relation between files and ℓ -objects are defined by the programmer by means of the file operators 'get' and 'put', which denote respectively, the transfer from, and transfer to, the specified file. The syntax of file operator declarations is given by:

$$\begin{array}{l} \text{fop } \underline{\text{get}} \text{ width } W \text{ bet } T_\ell \text{ and } T_u \text{ from } (F_1, F_2, \dots, F_n) \text{ to} \\ \hspace{20em} (L_1, L_2, \dots, L_m) \quad (4-23) \end{array}$$

$$\begin{array}{l} \text{fop } \underline{\text{put}} \text{ width } W \text{ beg } T_\ell \text{ and } T_u \text{ from } (L_1, L_2, \dots, L_m) \text{ to} \\ \hspace{20em} (F_1, F_2, \dots, F_n) \quad (4-24) \end{array}$$

The meaning of the width and time clauses are identical to those for operators. The list ' F_1, F_2, \dots, F_n ' denote names of previously declared files such that they form the source objects for the get declaration, and the sink objects for the put declaration. The list ' L_1, L_2, \dots, L_m ' denote a list of ℓ -objects which form the sink in (4-23) and the source in (4-24).

For example, assuming that the ℓ -objects 'INR1', 'INR2', 'OUTR1' and 'OUTR2' have been previously declared, then the declarations

```
fop get width 8 bet T1 and T1 from (CARDR01) to (INR1, INR2)
```

```
fop put width 8 bet T1 and T1 from (OUTR1, OUTR2) to
```

(CARDP10)

indicate that there exists transfer paths from the file 'CARDR01' to the ℓ -objects 'INR1' and 'INR2', and that transfer from 'CARDR01' to these ℓ -objects may be initiated within the time period denoted by 'T1' using the operator 'get'; similarly, there exists a path from both the ℓ -objects 'OUTR1' and 'OUTR2' to the file 'CARDP10', which may be activated using the operator 'put'.

CHAPTER V

EXECUTIONAL STATEMENTS

5.1 Preliminary Remarks

The executorial facilities in MICROL are represented by statements and procedures. The structure of these entities at the user level is similar to corresponding entities in ALGOL 60 [29] but there are important semantic distinctions because the concept of location-objects is specific to microprogramming. Furthermore, the validity of each statement in a MICROL program is defined by the data-path restrictions as specified by operator declarations. To indicate data-path restrictions, the following notation is used: The string

$$A \implies B \qquad (5-1)$$

denotes that a data-path exists from 'A' to 'B' where 'A' and 'B' are either ℓ -objects or files. This path exists if there is a declaration for one of the operators ':=', 'get' or 'put' with 'A' as the source and 'B' as the sink. Furthermore, if

$$A \implies B , \quad B \implies C , \quad C \implies D \qquad (5-2)$$

then an indirect data-path exists from 'A' to 'D'; this is denoted by

$$A \xrightarrow{*} D \quad (5-3)$$

There are four data types in MICROL, namely integers, characters, bit strings and the truth-values 'true' and 'false'. The content of an ℓ -object is interpreted as an integer when the ℓ -object appears in an arithmetic expression or a relation, and as a bit string when it is part of a boolean or status-test expression.

The content of a file cell is always interpreted as a char (character). The value of an expression is interpreted as an integer when it is an arithmetic expression, and as a truth-value if the expression is a relation or a status-test. The value of a boolean expression is always a bit string.

5.2 Expressions

Expressions in MICROL may be simple or subscripted ℓ -objects, simple arithmetic expressions (SAE), relations, status-tests, simple boolean expressions (SBE), conditional arithmetic expressions (CAE) or conditional boolean expression (CBE).

When the ℓ -object appears as an expression, its syntax is given by

$$L \quad (5-4)$$

$$L[S] \quad (5-5)$$

$$L[S_1::S_2] \quad (5-6)$$

where 'L' is the name of an ℓ -object, and 'S', 'S₁' and 'S₂' are subscripts which may be positive integers or simple ℓ -objects.

Eqn. (5-4) denotes a simple ℓ -object. Expressions of the form (5-5) are termed single-subscripted ℓ -objects, the subscript identifying a single element of the ℓ -object 'L'. When two subscripts are used as in (5-6), the first denotes the lower bound and the second, the upper bound of the elements being referenced in 'L'. For example, considering integer subscripts, and given the following declarations:

loc MBR = seq (8) bit (5-7)

loc REGARR = seq (4) MBR (5-8)

then the subscripted ℓ -object (5-9) references the fourth register in the register array, and (5-10) references bits 2 through 5 of 'MBR':

REGARR [4] (5-9)

MBR [2::5] (5-10)

When the subscripts in the expressions (5-5) and (5-6) are themselves simple ℓ -objects, then the expressions are semantically defined as follows: for a single subscripted ℓ -object 'L[S]', there must exist some other ℓ -object 'P' such that

$S \xrightarrow{*} P$

and 'P' points to 'L' by means of a pointer declaration. Similarly, for a double subscripted ℓ -object ' $L[S_1:S_2]$ ', there must exist ℓ -objects ' P_1 ' and ' P_2 ' such that

$$S_1 \xRightarrow{*} P_1$$

$$S_2 \xRightarrow{*} P_2$$

and ' P_1 ' and ' P_2 ' (which may be identical) both point to 'L'. It follows from the above that subscripted ℓ -objects in which the subscripts are themselves ℓ -objects are defined only when the referenced ℓ -object 'L' represents a random access memory.

As examples of ℓ -objects appearing as expressions, the following portion of a MICROL program is illustrative:

loc MAR = loc ACC = loc WORD = loc GPR = seq (8) bit

loc MEMORY = seq (256) WORD

⋮

op := width 8 bet ... and ... from (ACC) to (MAR)

op ptr width 0 bet 0 and 0 from (MAR) to (MEMORY)

Given these declarations, the following subscripted ℓ -objects are valid

ACC [2] (5-11)

GPR [3::6] (5-12)

MEMORY [MAR] (5-13)

MEMORY [ACC]

(5-14)

Expressions (5-11) and (5-12) are valid since both 'ACC' and 'GPR' are defined as 8-bit vectors, and the integer subscripts identify specific elements of these ℓ -objects. The expression (5-13) is valid since 'MAR' has been declared to point to 'MEMORY', while (5-14) is defined since

$$\text{ACC} \implies \text{MAR}$$

In contrast, the expression

ACC [WORD]

is undefined since there is no data-path from 'WORD' to any other ℓ -object pointing to 'ACC'.

Finally, it is important to note that for double-subscripted ℓ -objects, the two subscripts are either both positive integers or both simple ℓ -objects.

A simple arithmetic expression has the general syntactic form:

$$P_1 \ O_1 \ P_2 \ O_2 \ P_3 \dots\dots\dots P_{n-1} \ O_{n-1} \ P_n \quad (5-15)$$

where the P_i 's are either ℓ -objects or integer constants, and the O_i 's designate arithmetic operators. Simple arithmetic expressions describe a composite micro-operation with an integer domain and an integer result. The evaluation of an SAE proceeds from left to right and may be represented by the following sequence:

$$\begin{aligned}
 V &:= P_1 O_1 P_2 \\
 V &:= V O_2 P_3 \\
 V &:= V O_3 P_4 \\
 &\vdots \\
 V &:= V O_{n-1} P_n
 \end{aligned}$$

where 'V' is the ℓ -object specified on the left side of the assignment statement of which the expression (5-15) forms a part (see 5.3.1). Each element of the above sequence is defined only when it satisfies the required data-path conditions. For example, the first element of the sequence,

$$V := P_1 O_1 P_2$$

is defined only when there are data-paths from 'P₁' and 'P₂' to the source operands of the operator 'O₁', and from the operator's sink operand to 'V'.

A relation is of the syntactic form

$$V_1 R_1 V_2 \tag{5-16}$$

where 'V₁' and 'V₂' are ℓ -objects, referred to as the 'comparands' of 'R₁', and 'R₁' is a language-defined relational operator belonging to the set

$$R = \{=, \neq, <, >, \leq, \geq\} \tag{5-17}$$

A relation is a predicate which produces a truth-value 'true' or 'false' as a result of comparing two integer

values. The value of the relation is 'true' if the condition being tested is satisfied, otherwise it is 'false'.

Relational operators differ from the operators so far discussed in that they have no direct correspondence with the processor's operational units. MICROL relations are semantically defined only when their execution generates (in the microcode) a sequence of more primitive operations, for example assignment and arithmetic operations.

A status-test expression is also a predicate which produces a value 'true' or 'false'. The syntax of this expression is of the form

test V_1 (5-18)

or

test $V[S]$ (5-19)

where ' V_1 ' is an ℓ -object consisting of a single bit or is a status-object, and ' $V[S]$ ' is a single-subscripted ℓ -object with ' V ' as a bit sequence. In other words, test operates on a single bit.

If the value of the referenced bit at the time of evaluating this expression is '1', then the expression yields a 'true' value; if the referenced bit contains a '0', then the expression value is 'false'. Like relational operators, 'test' is a language-defined operator.

Boolean expressions describe computations with bit strings. A Boolean primary is of the syntactic form

$$P_1 \ B_1 \ P_2 \quad (5-20)$$

where ' P_1 ' and ' P_2 ' are ℓ -objects or binary string constants, and ' B_1 ' is a dyadic Boolean operator belonging to the set

$$D = \{\wedge, \vee, \oplus\} \quad (5-21)$$

A monadic Boolean expression possesses the syntax

$$(MP) \quad (5-22)$$

where ' P ' is an ℓ -object, and ' M ' an operator from the set

$$E = \{\neg, \underline{shl}, \underline{shr}, \underline{shli}, \underline{shri}, \underline{shlc}, \underline{shrc}\} \quad (5-23)$$

Finally, a simple Boolean expression is either a Boolean primary or a monadic Boolean expression, or a string of the form

$$S_1 \ B_1 \ P_1 \quad (5-24)$$

where ' S_1 ' is a simple Boolean expression, and ' P_1 ' is an ℓ -object or a binary constant.

The semantics of SBE's are similar to that for SAE's except that the domain and range of the expressions are bit strings. The evaluation of the SBE proceeds from left-to-right as in the case of SAE's.

Note that when a parenthesized monadic expression is encountered as an operand it is always the left-most element of the SBE. It follows that only one monadic expression may be present in an SBE. This restriction arises because a monadic expression, when encountered, has to be evaluated before the rest of the expression can be evaluated, regardless of the left-to-right rule. If the monadic expression is not the left-most element, then the results of its evaluation cannot be stored in the left hand ℓ -object, since the latter's current value would be destroyed. As a specific example, consider the statement (see 5.3):

$$L := A \wedge B \wedge (\underline{\text{shl}} C) \quad (5-25)$$

which contains an illegal SBE. The evaluation of (5-25) proceeds from left-to-right, and generates the sequence:

$$L := A \wedge B$$

$$L := L \wedge (\underline{\text{shl}} C)$$

At this stage, the evaluation of 'shl C' destroys the current value of 'L'.

A conditional expression (arithmetic or Boolean) is a means of selecting one of a given set of simple (arithmetic or Boolean) expressions on the basis of evaluating one or more predicates. The conditional arithmetic expression has the general syntactic form:

$$\begin{array}{l} \underline{\text{if}} \ R_1 \ \underline{\text{then}} \ A_1 \ \underline{\text{else}} \ \underline{\text{if}} \ R_2 \ \underline{\text{then}} \ A_2 \ \underline{\text{else}} \ \dots \ \underline{\text{else}} \\ \qquad \underline{\text{if}} \ R_n \ \underline{\text{then}} \ A_n \ \underline{\text{else}} \ A_0 \end{array} \quad (5-26)$$

where the quantities ' R_1 ', ' R_2 ', ..., ' R_n ' represent either relations or status-test expressions, and the ' A_i ' denote simple arithmetic expressions.

The value of a CAE is always an integer and is obtained according to the following rules:

- (a) Starting from ' R_1 ', evaluate the predicate ' R_i ';
- (b) If ' R_i ' has the value 'true', then the value of the CAE is the value of the SAE ' A_i ';
- (c) If ' R_i ' produces the value 'false', then steps (a) and (b) are repeated for ' R_{i+1} '. Finally, when the relation ' R_n ' produces a value 'false', then the value of the CAE is that of ' A_0 '.

The syntax and semantics of conditional Boolean expressions are defined analogously to those for CAE's, except that the alternative actions selected are SBE's. CBE's therefore yield bit strings as values. If ' B_i ' represents a simple Boolean expression, the syntax of CBE's is symbolized by:

$$\begin{array}{l} \underline{\text{if}} \ R_1 \ \underline{\text{then}} \ B_1 \ \underline{\text{else}} \ \underline{\text{if}} \ R_2 \ \underline{\text{then}} \ B_2 \ \underline{\text{else}} \ \dots \ \underline{\text{else}} \\ \qquad \underline{\text{if}} \ R_n \ \underline{\text{then}} \ B_n \ \underline{\text{else}} \ B_0 \end{array} \quad (5-27)$$

where as before, the ' R_i ' designate predicates.

5.3 Statements

The syntax of MICROL statements resembles that of most high-level software languages, in particular, ALGOL 60. However, the semantics differs because of the data-path restrictions existing in the processor. MICROL statements include assignment statements, goto statements, compound statements, conditional and for statements, procedure statements and declarations, and file statements.

Statements are MICROL representation of microprograms (or a part of a microprogram); each statement is mapped into a sequence of microinstructions in the control memory, each microinstruction possessing a unique control memory address. Except for interrupts or branches, MICROL statements are executed sequentially except when parallelism is permitted within the processor. The conditions for parallelism are discussed in Chapter VI.

5.3.1 Assignment Statements

The assignment statement serves to assign the computed value of an expression to an ℓ -object. Its syntax is given by

$$L_1, L_2, \dots, L_n := E \quad (5-28)$$

where ' L_j ' ($j = 1, 2, \dots, n$) is a simple or subscripted

ℓ -object and 'E' is a MICROL expression. If $n > 1$, then (5-28) is a multiple assignment statement, whilst if $n = 1$, it is termed a simple assignment statement.

The semantics of assignment statements have already been partly described in connection with the semantics of expressions (see 5.2). For example, the simple assignment statement

$$V := E \quad (5-29)$$

where 'V' is an ℓ -object, and 'E' an SAE of the form (5-15), will be executed in the manner described while discussing the semantics of SAE's. The evaluation of a multiple assignment statement initially involves the evaluation of the right-most ℓ -object ' L_n ' as a simple assignment statement. The subsequent assignments

$$\begin{array}{lcl} L_{n-1} & := & L_n \\ & \vdots & \\ L_2 & := & L_3 \\ L_1 & := & L_2 \end{array}$$

are then made, provided that

$$\begin{array}{lcl} L_n & \xRightarrow{*} & L_{n-1} \\ L_{n-1} & \xRightarrow{*} & L_{n-2} \\ & \vdots & \\ L_3 & \xRightarrow{*} & L_2 \\ L_2 & \xRightarrow{*} & L_1 \end{array}$$

5.3.2 Goto Statements

A statement may be prefixed by a label (an identifier) forming a labelled statement with the syntax:

L : S (5-30)

where 'L' is the label, and 'S' denotes the statement. In terms of the generated microcode, 'L' refers to the control memory address of the first microinstruction generated as a result of translating 'S'.

A goto statement has the syntactic form

goto L (5-31)

where 'L' denotes a label. Semantically, the goto statement is identical to that in ALGOL 60.

This statement differs from the other statements in MICROL in that it is an explicit command to the control to effect a change in the statement sequence. That is, its execution is independent of the structure and behaviour of the processor. Execution of the goto statement involves the transfer of the control memory address equivalent of 'L', to the control memory address-register, and a transfer of control to the sequence of microinstructions commencing at this new address. The timing requirements for this transfer of control is determined by the sequencing and address

generation scheme implemented for the specific control being microprogrammed [16,26].

5.3.3 Compound Statements and Blocks

The syntax and semantics of compound statements and blocks are similar to corresponding entities in ALGOL 60 [29]. The syntax of compound statements and blocks are, respectively:

$$\underline{\text{begin}} \ S_1; S_2; \dots ; S_n \ \underline{\text{end}} \quad (5-32)$$

$$\underline{\text{begin}} \ D_1; D_2; \dots ; D_m; S_1; S_2; \dots ; S_n \ \underline{\text{end}} \quad (5-33)$$

Here the ' D_i 's' represent declarations, and the ' S_i 's' denote statements.

5.3.4 Conditional Statements

Conditional statements in MICROL are if-statements and if-then-else-statements, which are given respectively by the following syntax:

$$\underline{\text{if}} \ P \ \text{then} \ S_1 \quad (5-34)$$

$$\underline{\text{if}} \ P \ \underline{\text{then}} \ U \ \underline{\text{else}} \ S_2 \quad (5-35)$$

where ' P ' denotes a predicate yielding a truth-value, ' S_1 ' is any statement other than a conditional statement, ' U ' denotes any statement other than a conditional

or a for-statement (see 5.3.5), and ' S_2 ' is any statement other than an if-statement.

The execution of an if statement involves the evaluation of the predicate ' P ' and execution of ' S ' if the value of ' P ' is 'true'; otherwise, control is transferred to the statement immediately following the if statement.

If a jump is made from outside ' S ' to a labelled statement contained in ' S ' then the 'if P ' clause is disregarded. It is to be noted again that a predicate other than a relation or a status-test expression is illegal in MICROL. For example, a Boolean expression may not be used as a predicate in the if statement.

The if-then-else statement permits the selection and execution of one of several unconditional statements. Semantically, it is similar to the if statement: if the value of ' P ' in (5-35) is 'true', the statement ' U ' is executed, otherwise ' S_2 ' is executed. If ' S_2 ' is itself an if-then-else statement, then the following is obtained:

$$\underline{\text{if } P \text{ then } U \text{ else if } P_1 \text{ then } U_1 \text{ else } S_3} \quad (5-36)$$

The statements ' U ' and ' U_1 ' in (5-36) are referred to as the alternatives of the if-then-else statement, and the execution of (5-36) results in the alternative corresponding to the first 'true' predicate to be

executed. If none of the predicates are 'true', then ' S_3 ' is selected for execution.

A jump to one of the alternatives from outside the statement is illegal. However, if a jump from outside ' S_3 ' is made to a label contained in ' S_3 ', then the preceding part of the if-then-else statement is ignored.

5.3.5 Iterative Statements

Microprogrammed iterations may be of the following types:

(a) A sequence of microinstructions are iteratively executed until an S-object is set 'on' or 'off' by virtue of one or more of the iterated instructions. The state of the S-object is altered by the hardware as a consequence of the operations performed within the iteration. However, the S-object state (value) has to be explicitly tested by the microprogram in each iteration.

Iterations of this type are described in MICROL by conditional statements, for example

```
L1: if test OVFL0 then goto EXIT else S; goto L1;
```

where 'OVFL0' is the S-object controlling the execution of 'S'. Exit from the loop occurs when the 'test OVFL0' expression yields a 'true' value.

(b) An ℓ -object which acts as a counter, is explicitly incremented (or decremented) from an initial value by the programmer in each iteration. Control passes from the loop when the counter contains a desired value.

MICROL permits the specification of iterations of this type by means of the for-statement which has the following syntactic form:

$$\underline{\text{for}}\ V\ \underline{\text{from}}\ X_1\ \underline{\text{step}}\ X_2\ \underline{\text{until}}\ X_3\ \underline{\text{do}}\ S \quad (5-37)$$

where 'V' represents an ℓ -object, ' X_1 ', ' X_2 ', and ' X_3 ' are either ℓ -objects or integer constants, and 'S' is either an assignment statement or a compound statement.

The semantics of MICROL for-statements are identical to those for the corresponding construct in ALGOL 60 [29]; in addition, the following data-path conditions must be satisfied:

If ' X_1 ' is an ℓ -object, then

$$X_1 \xRightarrow{*} V$$

since the assignment ' $V := X_1$ ' is made during the execution of (5-37). Furthermore, since the value of 'V' is increased in each iteration by the value of ' X_2 ', in effect the following statement is also generated:

$$V := V + X_2 \quad (5-38)$$

Therefore, if ' X_2 ' is an ℓ -object, then 'V' and ' X_2 ' must satisfy the data-path conditions

$$V \xrightarrow{*} IN3$$

$$X_2 \xrightarrow{*} IN4$$

$$OUT1 \xrightarrow{*} V$$

where 'IN3' and 'IN4' are declared as source operands to the '+' operator, and 'OUT1' has been defined as its result operand.

It should be noted that iterations of both types above may also be described by means of 'while-statements' of the form

while R do S

where 'R' is a relation, and 'S', the statement(s) to be executed. In designing MICROL however, the for-statement was chosen for specifying iterative processes because of its association with ALGOL 60. Furthermore, the while-statement suffers from two disadvantages. Firstly, considering iterations of type (a) above, a while-statement corresponding to the given example would have to be of the form:

while OVFL0 = 0 do S

However, 'OVFL0' is a status-object so that it cannot participate in a relational expression (eqn. 5-16). This statement is thus undefined. A possible solution is to provide an additional construct of the form

do S until test OVFL0

Secondly, for iterations of type (b), the use of the while-statement would require the following set of statements:

V := X₁;

while X₁ ≤ X₃ do begin S ; X₁ := X₁ + X₂; end

That is, additional initializing and incrementing statements are necessary.

5.3.6 File Transfer Statements

File transfer statements are commands for transferring data between files and ℓ-objects. They are analogous to assignment statements, and have the following syntax:

get F to V₁, V₂, , V_n (5-39)

put V₁ into F (5-40)

where the 'V_i's' denote ℓ-objects, and 'F' denotes a file name. The get statement (5-39) involves the transfer of a char from the currently active cell in the file 'F' to each of the ℓ-objects 'V₁', 'V₂', 'V_n'. This process is executed by first transferring from 'F' to 'V₁' and subsequently from 'V₁' to 'V₂',

'V₁' to 'V₃' and so on. This implies that in order that (5-39) be defined, the following conditions have to be satisfied:

$$F \xRightarrow{*} V_1$$

$$V_1 \xRightarrow{*} V_2$$

$$V_1 \xRightarrow{*} V_3$$

$$\vdots$$

$$V_1 \xRightarrow{*} V_n$$

The put-statement (5-40) is executed by transferring the contents of the ℓ -object 'V₁' to the active cell of 'F'. The statement is executed correctly only if

$$V_1 \xRightarrow{*} F$$

Furthermore, execution of the put statement requires that the bit string contained in 'V' when encoded before transferring to the file 'F', yield a valid 'char'.

At the end of statement execution, the currently active cell of the referenced file is deactivated, and the next cell is made active. Thus if two consecutive get (or put) statements are executed, the result is that two consecutive chars from consecutive cells are read from (or written into) respectively.

It is important to note that the execution of file transfer statements results in the actual transfer between I/O devices and memories. For processors in

which I/O transfers are executed by separate data channels and if the channel commands are executed under microprogram control, then the file transfer statements will be part of the channel microprograms. The microprograms may reside in the main processor's control memory as in the System/360 Models 40 and 50 [16]. Alternatively, if the channels are separate processors, they may possess their own control memories which will then contain the channel microprograms.

5.3.7 File Control Statements

A set of language-defined primitive control operators are available whereby the MICROL programmer may issue file control statements. The file control operators are as follows:

init : its action is to initiate and make active, a composite file unit (e.g. a card or a tape record) for subsequent file operations.

term : the converse of the init operator, its effect is to terminate the activity or complete the activity of some composite unit of a file.

bsp : defined only for files identifying magnetic tapes; this operator deactivates the current cell and activates the preceding cell of the designated file.

rwd : also defined only for files identifying magnetic tapes, its action is to cause continuous

backspace of the designated file until the first cell of the first record becomes active.

The init and term statements possess the following syntactic forms:

init F (5-41)

init F₁[V] (5-42)

term F₂ (5-43)

where 'F' denotes any file that is not of the form 'diski' or 'drumj'; 'F₁' symbolizes files of the form 'diski' or 'drumj' only; 'V' denotes a simple l-object, and 'F₂' denotes any file. The semantics of the initiating and terminating statements depend on the precise class of files addressed by them. For the init statement, the semantics are defined as follows:

(a) If 'F' is of the form 'cardri', then the statement causes a card to be made available for some subsequent file operation. In this instance, the composite file unit is a card. The first cell of the file (i.e. the first column) is made active. Similarly if 'F' is of the form 'cardpi', the statement causes a blank card to be made available. The first cell of the file is put into the active state, and depending on the implementation, this may correspond to either a column or a row.

(b) When 'F' refers to a magnetic tape, execution of (5-41) results in the next inter-record gap being sensed by the read-write head. That is, the next tape record is made available for subsequent file operations, and the first cell of this record activated. The composite unit for this file is a record. The init statement for a magnetic tape file is defined only when 'F' has been previously declared within a get declaration as a source operand. Otherwise, if 'F' was declared as a sink operand, then the file would be defined for output operations only, so that if it were a blank tape, the inter-record gap would never be sensed.

(c) If 'F' is of the form 'linei', then the statement execution results in the next line of printing to be made available. The composite unit for this file is a line. The first cell of the line is activated.

(d) When 'F' denotes a teletype reader or printer, the init statement causes a carriage return action, a new line of char or a line of printing to be made available, and the first cell of the line to be activated.

(e) If 'F' is of the form 'ptaperi' the initiating statement results in the next record separator character to be sensed. That is, a record on the paper tape designated by the file name is ready for subsequent file operations, with the first cell in the active

state. An init statement for a file designating a paper tape punch is undefined since the file denotes a blank paper tape and the record separator would never be sensed.

(f) For files designating random-access devices, the init statement of the form (5-42) is used, in which 'V' denotes an ℓ -object containing the address of some composite file unit. That is, 'V' designates a file address. The result of executing the init statement is to ready a track, or a track and sector (depending on the file organization), as determined by the contents of 'V', and activate the first cell of the composite unit. The statement is only defined if there is some ℓ -object 'P' such that

$$V \xRightarrow{*} P$$

and 'P' is declared to point to the file 'F₁'.

The term statement (5-43) is only defined for files of the type 'ptapep' and 'mtape'. When 'F₂' denotes a paper tape punch, the statement causes a record separator character to be punched. Similarly, for magnetic tapes, an inter-record gap is created on the designated file, provided that 'F₂' is declared as a sink within a put declaration.

There are two special file control statements defined in MICROL for files denoting 'mtape'. The

syntax for these statements are as follows:

bsp F (5-44)

rwd F (5-45)

Execution of (5-44) causes the currently active cell of the file 'F' to be de-activated and the preceding cell to be made active. The rwd statement causes a sequence of bsp operations to be performed in succession until the first cell of 'F' is under the read-write head.

5.4 Procedures

The concept and use of subroutines is provided in MICROL in the form of procedures, which may be called by procedure statements located elsewhere in the micro-program. Syntactically, MICROL procedures are similar to those in ALGOL 60 [29]. However, for procedures with parameters, all the parameters are called by value prior to executing the procedure. Furthermore, the parameter mechanism in MICROL procedures is considerably simpler.

5.4.1 Procedure Declarations

The syntax of procedure declarations are given by:

proc P B (5-46)

proc P(V_1, V_2, \dots, V_n) B (5-47)

where 'P' denotes the procedure name, ' (V_1, V_2, \dots, V_n) ' is the list of formal parameters, where each ' V_i ' is an ℓ -object or a file, and 'B' is the procedure body which may be either a compound statement or a block. The declaration (5-46) is termed a parameterless procedure.

A MICROL procedure declaration represents a micro-programmed subroutine and may itself contain within its body, a call to another procedure. The number of procedures that may be nested will be a function of the language implementation.

A procedure declaration is equivalent to a block, so that its execution implies essentially the execution of this 'block', and the operands of the procedure - that is, the quantities involved in its execution - are the 'operands' of this block. Operands may be 'global' or 'formal' as in ALGOL 60 [29].

Since the semantics of procedure declarations are closely connected to procedure calls, a discussion of the semantics is deferred to section 5.4.3.

5.4.2 Procedure Statements

A procedure statement serves to initiate execution of a declared procedure. The statement has one of the following syntactic forms:

call P (5-48)

call P(V_1, V_2, \dots, V_n) (5-49)

where 'P' denotes the name of the procedure being called, and ' (V_1, V_2, \dots, V_n) ' denotes a list of actual parameters, defining the ℓ -objects or files that the procedure uses.

The form of a specific procedure statement is restricted by the form of the corresponding procedure declaration. Thus if 'P' represents the procedure name, then the statement must be of the form (5-48) if the corresponding declaration for 'P' is of the form (5-46); similarly the statement is of the form (5-49) if the corresponding declaration is given by (5-47). Furthermore, the length of the formal and actual parameter lists must be equal.

All identifiers occurring in a procedure statement must represent quantities that are defined when the statement is executed. Thus the statement must reside in some block that is identical to, or within, a block containing the procedure declaration, so that the name 'P' satisfies this condition. The actual parameters ' V_i ' must also have been declared as ℓ -objects or as files in some block that is identical to, or external to, the block containing the procedure declaration, so that when control is transferred from the procedure statement, objects corresponding to the actual parameters exist in the procedure's environment.

5.4.3 Semantics of MICROL Procedures

When a procedure is called from somewhere in the program, the corresponding declaration becomes 'active'. However, the execution of the procedure is valid only when the following conditions are satisfied:

(a) Given the formal parameter list ' (F_1, F_2, \dots, F_n) ' in the declaration, and the actual parameter list ' (A_1, A_2, \dots, A_n) ' in the statement, there must exist the data-paths

$$A_i \xRightarrow{*} F_i \quad i = 1, 2, \dots, n \quad .$$

If such data-paths exist, then the contents of each ' A_i ' is transferred to the corresponding ' F_i ' at the time of the procedure call, and before the procedure is executed. That is, for procedures with parameters it is the value contained in the ' A_i 's' at the time of the call that are used in executing the procedure. If ' A_i ' is the file name, the value used is the internal representation of the 'char' contained in the currently active cell of the designated file.

(b) When global operands are encountered, the values of the operands are those that were contained at the time of entry to the procedure declaration if the declaration were to be regarded as a block. That is, the values of the global operands are those values contained in them when the declaration is first encountered.

If the operand is a file, then the corresponding value would be that contained in the cell that was active at the time of encountering the procedure declaration.

CHAPTER VI

SUMMARY AND RECOMMENDATIONS

6.1 A Summary of the Capabilities of MICROL

It is felt by this author that previous work on high-level microprogramming languages has been rather arbitrary: that the currently available languages were developed without adequately establishing a logical, a priori framework. This consideration motivated the theoretical framework developed in Chapter III. It is felt that the use of this framework in defining MICROL resulted in a more logical and unambiguous language than those available at present.

The main features of MICROL are summarized as follows:

- (a) The syntax of MICROL executional statements are based on ALGOL 60.
- (b) Variables in MICROL statements are not data-objects of fixed 'types' but are location-objects, files, and status-objects. The values of these entities are interpreted as integers, bit strings, truth values, or characters according to the operations performed on them.
- (c) A set of composition rules are provided whereby primitive ℓ -objects can be combined to form more complex ℓ -objects. This permits emulated memory elements

to be defined in terms of the 'host' processor's memory elements.

(d) A set of language-defined operators is used in the form of operator declarations to describe the processor's data-flow. These operators are MICROL representations of hardware operational units.

(e) File-statements and file-control-statements provide facilities for microprogramming I/O operations.

(f) The provision of time-object declarations.

MICROL extends the capabilities of the languages described in Chapter II in the following ways:

(a) A set of declarative facilities is provided by which a large class of processor organizations can be defined within a MICROL program. Furthermore, the language may be used to microprogram both horizontal and vertical control stores. In this respect, it combines the capabilities of Husson's language, MPL, and CDL.

(b) A common characteristic of the languages described in Chapter II is the lack of facilities for I/O operations. These facilities are provided in MICROL.

(c) The availability of time-object declarations allow the timing-attributes of the processor to be defined. This facility is provided in CDL, but to a very limited extent. MICROL allows a wider class of timing attributes, both for memories and operational units, to be specified.

6.2 Further Research

As noted in Chapter I, the scope of this thesis excluded the subject of implementation. The most obvious extension of the present work would be the design of a MICROL compiler. In particular, the following aspects of the compilation process should be investigated:

(a) The syntax of MICROL has been defined by means of a simple precedence grammar [1], and is given in Appendix A. The choice of this grammar was motivated primarily by the fact that a simple precedence grammar generates an unambiguous language [1]. It is asserted therefore that MICROL is unambiguous.

The simple precedence grammar is relatively easy to construct, and has been widely used for other programming languages [38,39]. Unfortunately, the requirement that pairs of symbols must have unique precedence relation between them, necessitates a large number of additional production rules. For purposes of implementation, a more efficient grammar, such as the LR(k) type [1] should be developed.

(b) MICROL has been defined using ALGOL 60 as a basis, and many of the concepts employed in constructing ALGOL compilers may possibly be used in implementing MICROL. The main departure from ALGOL implementation

would be in the code generation routine, since the assignment of physical resources (main memory words, local store, and hardware registers) to ℓ -objects is actually determined by MICROL declarative statements. Furthermore, the timing attributes of operational units have to be used in order to determine the time-validities of the resulting micro-operations. Investigations of efficient code generation procedures for MICROL are therefore suggested.

(c) The compilation strategy should be such as to provide maximum emphasis on optimizing the microcode at the expense of, if necessary a slow compiler. The latter would not be a serious disadvantage, since the total compilation time for any set of MICROL programs should be very much smaller than the total execution time for the resulting microcode.

Microprogram optimization was first studied by Kleir and Ramamoorthy [21], and more recently, by Sitton [31]. In essence, microcode optimization involves the identification of alternative and equivalent micro-operation ('action') forms; the analysis of these equivalent action forms in order to identify redundant actions; and the construction of a set of deletion strategies for the optimal deletion of redundant actions. Further optimization studies can proceed in two directions: firstly, the implementation of the

algorithms developed in [21,31] in practical compilers; and secondly, the development of more general strategies for the optimizing microcode.

(d) Another facet to microprogram optimization relates to horizontal microprograms, in which several micro-operations may be placed in the same microinstruction word: that is, parallel micro-operations are possible. The conditions for parallelism in microprograms are rather more complex than those established for multiprocessing [2,4]. An algorithm for identifying parallel micro-operations is presented in [8]. Further investigations in this area are necessary, particularly with regard to the implementation of this algorithm within the framework of real compilers, and the development of alternate algorithms.

REFERENCES

- (1) AHO, A.V. and ULLMAN, J.D., The Theory of Parsing, Translation, and Compiling, Vol.I: Parsing, Prentice-Hall, Englewood-Cliffs, N.J., 1973.
- (2) BAER, J.L., "A Survey of Some Theoretical Aspects of Multiprocessing", ACM Computing Surveys, Vol. 15, #1, pp.31-80, March 1973.
- (3) BELL, C.G. and NEWELL, A., Computer Structures: Readings and Examples, McGraw-Hill, N.Y., 1970.
- (4) BERNSTEIN, A.J., "Analysis of Programs for Parallel Processing", IEEE Trans. on Comput., Vol. C-15, #5, pp.757-763, Oct. 1966.
- (5) CHU, Y., "An ALGOL-like Computer Design Language", Comm. ACM., Vol. 8, #10, pp.607-615, Oct. 1965.
- (6) CHU, Y., Computer Organization and Microprogramming, Prentice-Hall, Englewood-Cliffs, N.J., 1972.
- (7) COOK, R.W. and FLYNN, M.J., "System Design of a Dynamic Microprocessor", IEEE Trans. on Comput., Vol. C-19, #3, pp.213-222, March 1970.
- (8) DASGUPTA, S. and JACKSON, L.W., An Algorithm for Identifying Parallel Micro-operations, Tech. Rept. TR 73-20, Dept. of Computing Science, University of Alberta, Edmonton, Canada, Dec. 1973.
- (9) DONALDSON, R.G., Implications of Microprogramming, M.Sc. Thesis, Dept. of Computing Science, University of Alberta, Edmonton, Canada, 1973.

- (10) ECKHOUSE, R.H., "A High-Level Microprogramming Language (MPL)", SJCC AFIPS Conf. Proc., Vol. 38, pp.169-177, 1971.
- (11) FALKOFF, A.D., IVERSON, K.E. and SUSSENGUTH, E.H., "A Formal Description of System/360", IBM Systems J., Vol. 3, #3, pp.198-262, 1964.
- (12) GLUSHKOV, V., "Automata Theory and Formal Microprogram Transformations", Kibernetika, Vol. 1, #5, 1965.
- (13) HALE, S.J., "Emulator Support for High-Level Languages", Proc. 3rd ACM Workshop on Microprogramming, Oct. 1970.
- (14) HATTORI, M., YANO, M., and FUJINO, K., "MPGS: A High-Level Language for Microprogram Generating Systems", Proc. ACM Annual Conf., pp.572-581, Aug. 1972.
- (15) HELLERMAN, H., Digital Computer Systems Principles, McGraw-Hill, N.Y., 1967.
- (16) HUSSON, S.S., Microprogramming: Principles and Practices, Prentice-Hall, Englewood-Cliffs, N.J., 1970.
- (17) ITO, T., "A Theory of Formal Microprograms", SIGMICRO Newsletter, Vol. 4, #1, pp.5-17, April 1973.
- (18) IVERSON, K.E., A Programming Language, John Wiley and Sons, N.Y., 1962.
- (19) IVERSON, K.E. and BROOKS, F.P., Automatic Data Processing, System/360 Edition, John Wiley and Sons, N.Y., 1969.

- (20) JAKOLAT, F.A., "Advantages of Large Microprogram Control Words", SIGMICRO Newsletter, Vol. 2, #3, pp.15-17, Oct. 1971.
- (21) KLEIR, R.L. and RAMAMOORTHY, C.V., "Optimization Strategies for Microprograms", IEEE Trans. on Comput., Vol. C-20, #7, pp.783-795, July 1971.
- (22) MAGILBY, K.B., "System Organization Considerations for Microprogrammed Processors", SIGMICRO Newsletter, Vol. 2, #3, pp.11-15, Oct. 1971.
- (23) NAUR, P. (Editor), "Revised Report on the Algorithmic Language ALGOL 60", Comm. ACM, Vol. 6, pp.1-17, Jan. 1963.
- (24) NEWTON, G.E., "A Microprogrammed Parser for a Subset of APL", Proc. 3rd Workshop on Microprogramming, Oct. 1970.
- (25) RAUSCHER, T.G. and AGARWALA, A.K., "On the Syntax and Semantics of Horizontal Microprogramming Languages", Proc. ACM Annual Conf., pp.52-56, Aug. 1973.
- (26) REDFIELD, S.R., "A Study in Microprogrammed Processors: A Medium-Sized Microprogrammed Processor", IEEE Trans. on Comput., Vol. C-20, #7, pp.743-750, July 1971.
- (27) ROSIN, R.F., "Contemporary Concepts of Microprogramming and Emulation", ACM Computing Surveys, Vol. 1, #4, pp.197-212, Dec. 1969.
- (28) ROSIN, R.F., FRIEDER, G. and ECKHOUSE, R.H., "An Environment for Research in Microprogramming and Emulation", Comm.ACM, Vol.15, #8, pp.748-760, Aug. 1972.

- (29) RUTHAUSER, H., Description of ALGOL 60, Springer-Verlag, Berlin, 1967.
- (30) SHRIVER, B.D., "Microprogramming and Numerical Analysis", IEEE Trans. on Comput., Vol. C-20, #7, pp. 808-811, July 1971.
- (31) SITTON, W.G., Strategies for Microprogram Optimization, Tech. Rept. TR73-4, Dept. of Computing Science, University of Alberta, Edmonton, Canada, April 1973 (Ph.D. Thesis).
- (32) TUCKER, A.B. and FLYNN, M.J., "Dynamic Microprogramming: Processor Organization and Programming", Comm. ACM., Vol. 14, #4, pp.240-250, April 1971.
- (33) WERHEISER, A.H., "Microprogrammed Operating Systems", Proc. 3rd ACM Workshop on Microprogramming, Oct. 1970.
- (34) WILKES, M.V., "The Best Way to Design an Automatic Calculating Machine", Manchester Univ. Computer Inaug. Conf. 1951.
- (35) WILKES, M.V. and STRINGER, J.B., "Microprogramming and the Design of Control Circuits in an Electronic Digital Computer", Proc. Camb. Phil. Soc., Vol. 49, 1953.
- (36) WILKES, M.V., "The Growth of Interest in Microprogramming - A Literature Survey", ACM Computing Surveys, Vol. 1, #3, pp.139-145, Sept. 1969.

- (37) WILKES, M.V., "The Use of a Writeable Control Memory in a Multiprogramming Environment", Proc. 5th ACM Workshop on Microprogramming, Urbana, Illinois, Sept. 1972, pp.62-65.
- (38) WIRTH, N., "PL 360 - A Programming Language for the 360 Computers", J. ACM., Vol. 15, #1, pp.37-74, Jan. 1968.
- (39) WIRTH, N. and WEBER, H., "Euler: A Generalization of ALGOL and its Formal Definition: Part II", Comm. ACM., Vol. 9, #2, pp.89-99, Jan. 1966.
- (40) YOUNG, S., "A Microprogram Simulator", SIGMICRO Newsletter, Vol. 2, #3, pp.43-56, Oct. 1971.
- (41) ———, Microprogramming Handbook, Microdata Corp., Santa Ana, Calif., Nov. 1971 (2nd Edition).

APPENDIX A

THE SYNTAX OF MICROL IN SIMPLE PRECEDENCE FORM

I. Declarations

- (1) $\langle \ell\text{-obj-decl} \rangle ::= \langle \text{simp-}\ell\text{-obj-decl} \rangle \mid \langle \text{mult-}\ell\text{-obj-decl} \rangle$
- (2) $\langle \text{simp-}\ell\text{-obj-decl} \rangle ::= \langle \ell\text{-symb-cl} \rangle \underline{\text{bit}} \mid \langle \ell\text{-symb-cl} \rangle \langle \text{char-string} \rangle \mid$
 $\langle \ell\text{-symb-cl} \rangle \langle \text{char-string} \rangle [\langle \text{identifier} \rangle] \mid$
 $\langle \ell\text{-symb-cl} \rangle \langle \text{char-string} \rangle [\langle \text{int-width} \rangle] \mid$
 $\langle \ell\text{-symb-cl} \rangle \langle \text{comp-}\ell\text{-obj} \rangle$
- (3) $\langle \text{mult-}\ell\text{-obj} \rangle ::= \langle \ell\text{-symb-cl} \rangle \langle \text{simp-}\ell\text{-obj-decl} \rangle \mid$
 $\langle \ell\text{-symb-cl} \rangle \langle \text{mult-}\ell\text{-obj-decl} \rangle$
- (4) $\langle \ell\text{-symb-cl} \rangle ::= \underline{\text{loc}} \langle \text{char-string} \rangle =$
- (5) $\langle \text{comp-}\ell\text{-obj} \rangle ::= \langle \text{seq-}\ell\text{-obj} \rangle \mid \langle \text{tuple-}\ell\text{-obj} \rangle \mid \langle \text{union-}\ell\text{-obj} \rangle$
- (6) $\langle \text{char-string} \rangle ::= \langle \text{identifier} \rangle$
- (7) $\langle \text{seq-}\ell\text{-obj} \rangle ::= \underline{\text{seq}}(\langle \text{int-width} \rangle) \underline{\text{bit}} \mid \underline{\text{seq}}(\langle \text{int-width} \rangle)$
 $\langle \text{letter-string} \rangle$
- (8) $\langle \text{tuple-}\ell\text{-obj} \rangle ::= \langle \text{phase-clause} \rangle$
- (9) $\langle \text{phase-clause} \rangle ::= (\langle \text{phase-list} \rangle)$
- (10) $\langle \text{phase-list} \rangle ::= \langle \text{phase-id-list} \rangle$
- (11) $\langle \text{phase-id-list} \rangle ::= \langle \text{letter-string} \rangle \mid \langle \text{letter-string} \rangle,$
 $\langle \text{phase-id-list} \rangle$

- ```

(12) <letter-string> ::= <char-string>

(13) <int-width> ::= <integer>

(14) <union-ℓ-obj> ::= union<tuple-ℓ-obj>

(15) <identifier> ::= <letter>|<identifier><letter>|
 <identifier><letter><digit>

(16) <letter> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|
 w|x|y|z|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|
 S|T|U|V|W|X|Y|Z

(17) <integer> ::= <digit>|0<integer>|1<integer>|2<integer>|
 3<integer>|4<integer>|5<integer>|6<integer>|
 7<integer>|8<integer>|9<integer>

(18) <digit> ::= 0|1|2|3|4|5|6|7|8|9

(19) <time-unit-decl> ::= <unit-symb-cl><unit-time-cl>

(20) <unit-symb-cl> ::= timeunit<char-string> =

(21) <unit-type-cl> ::= <int-width>nanosec

(22) <phase-decl> ::= <phase-symb-cl><phase-val-cl>

(23) <phase-symb-cl> ::= phase<char-string> =

(24) <phase-val-cl> ::= (<int-width>:<int-width>)<letter-string>
 (<int-width>:<int-width>) nanosec

(25) <bmc-decl> ::= <bmc-symb-cl><phase-clause>

```





- (26) <bmc-symb-cl> ::= bmc<char-string>=
- (27) <stc-decl> ::= <stc-symb-cl><phase-clause>
- (28) <stc-symb-cl> ::= stc<char-string>=
- (29) <rphase-decl> ::= <rphase-symb-cl><phase-cl>
- (30) <rphase-symb-cl> ::= rphase<char-string>=
- (31) <wphase-decl> ::= <wphase-symb-cl><phase-cl>
- (32) <wphase-symb-cl> ::= wphase<char-string>=
- (33) <phase-rel-decl> ::= <phase-rel-arg><phase-rel-funct>
- (34) <phase-rel-arg> ::= sync<char-string>
- (35) <phase-rel-funct> ::= with<char-string>|<with int-width>
- (36) <file-decl> ::= <file-name-cl><status-obj-cl>
- (37) <file-name-cl> ::= file<file-name>
- (38) <status-obj-cl> ::= with<phase-clause>
- (39) <file-name> ::= <file-prefix><integer>
- (40) <file-prefix> ::= cardr|<cardp>|<mtape>|<disk>|<drum>|<line>|  
ttr|<ttp>|<ptaper>|<ptapep>
- (41) <operator-decl> ::= <op-symb-cl><op-width-cl>  
<op-time-cl><op-param-cl>
- (42) <op-symb-cl> ::= op<operator-list>



- (43)  $\langle \text{operator-list} \rangle ::= \langle \text{prim-op} \rangle | \langle \text{prim-op} \rangle, \langle \text{operator-list} \rangle$
- (44)  $\langle \text{prim-op} \rangle ::= + | - | \underline{\text{add}} | \underline{\text{sub}} | \underline{\text{plus}} | \underline{\text{min}} | \wedge | \vee | \neg | \oplus | \underline{\text{shl}} | \underline{\text{shr}} |$   
 $\underline{\text{shli}} | \underline{\text{shri}} | \underline{\text{shlc}} | \underline{\text{shrc}} | := | \underline{\text{ptr}}$
- (45)  $\langle \text{op-width-cl} \rangle ::= \underline{\text{width}} \langle \text{int-width} \rangle$
- (46)  $\langle \text{op-time-cl} \rangle ::= \langle \text{op-time-lower-bound} \rangle \langle \text{op-time-upper-bound} \rangle$
- (47)  $\langle \text{op-time-lower-bound} \rangle ::= \underline{\text{bet}} \langle \text{char-string} \rangle | \underline{\text{bet}} [ \langle \text{int-width} \rangle ]$
- (48)  $\langle \text{op-time-upper-bound} \rangle ::= \underline{\text{and}} \langle \text{char-string} \rangle | \underline{\text{and}} [ \langle \text{int-width} \rangle ]$
- (49)  $\langle \text{op-param-cl} \rangle ::= \langle \text{source-set} \rangle \langle \text{sink-set} \rangle$
- (50)  $\langle \text{source-set} \rangle ::= \underline{\text{from}} (\langle \text{phase-list} \rangle) : (\langle \text{phase-list} \rangle) |$   
 $\underline{\text{from}} (\langle \text{phase-list} \rangle)$
- (51)  $\langle \text{sink-set} \rangle ::= \underline{\text{to}} (\langle \text{phase-list} \rangle) : (\langle \text{phase-list} \rangle) |$   
 $\underline{\text{to}} (\langle \text{phase-list} \rangle)$
- (52)  $\langle \text{file-op-decl} \rangle ::= \langle \text{file-get-decl} \rangle | \langle \text{file-put-decl} \rangle$
- (53)  $\langle \text{file-get-decl} \rangle ::= \underline{\text{fop}} \underline{\text{get}} \langle \text{op-width-cl} \rangle \langle \text{op-time-cl} \rangle$   
 $\langle \text{get-param-cl} \rangle$
- (54)  $\langle \text{file-put-decl} \rangle ::= \underline{\text{fop}} \underline{\text{put}} \langle \text{op-width-cl} \rangle \langle \text{op-time-cl} \rangle$   
 $\langle \text{put-param-cl} \rangle$
- (55)  $\langle \text{get-param-cl} \rangle ::= \langle \text{get-source-set} \rangle \langle \text{get-sink-set} \rangle$
- (56)  $\langle \text{get-source-set} \rangle ::= \underline{\text{from}} (\langle \text{file-name-list} \rangle)$
- (57)  $\langle \text{get-sink-set} \rangle ::= \underline{\text{to}} (\langle \text{phase-list} \rangle)$



- (58)  $\langle \text{file-name-list} \rangle ::= \langle \text{file-name} \rangle | \langle \text{file-name} \rangle ,$   
 $\langle \text{file-name-list} \rangle$
- (59)  $\langle \text{put-param-cl} \rangle ::= \langle \text{put-source-set} \rangle \langle \text{put-sink-set} \rangle$
- (60)  $\langle \text{put-source-set} \rangle ::= \text{from}(\langle \text{phase-list} \rangle)$
- (61)  $\langle \text{put-sink-set} \rangle ::= \text{to}(\langle \text{file-name-list} \rangle)$

## II. Statements

- (62)  $\langle \text{statement} \rangle ::= \langle \text{labelled-st} \rangle | \langle \text{unlabelled-st} \rangle$
- (63)  $\langle \text{labelled-st} \rangle ::= \langle \text{identifier} \rangle : \langle \text{unlabelled-st} \rangle$
- (64)  $\langle \text{unlabelled-st} \rangle ::= \langle \text{assign-st} \rangle | \langle \text{goto-st} \rangle | \langle \text{pro-st} \rangle |$   
 $\langle \text{cond-st} \rangle | \langle \text{for-st} \rangle | \langle \text{block} \rangle |$   
 $\langle \text{file-st} \rangle | \langle \text{file-ctl-st} \rangle$
- (65)  $\langle \text{assign-st} \rangle ::= \langle \ell\text{-obj-list} \rangle \langle \text{right-hand-expr} \rangle$
- (66)  $\langle \ell\text{-obj-list} \rangle ::= \langle \ell\text{-obj} \rangle | \langle \ell\text{-obj-list} \rangle , \langle \ell\text{-obj} \rangle$
- (67)  $\langle \ell\text{-obj} \rangle ::= \langle \text{alpha-string} \rangle | \langle \text{subscr-}\ell\text{-obj} \rangle$
- (68)  $\langle \text{alpha-string} \rangle ::= \langle \text{simp-}\ell\text{-obj} \rangle$
- (69)  $\langle \text{simp-}\ell\text{-obj} \rangle ::= \langle \text{identifier} \rangle$
- (70)  $\langle \text{subscr-}\ell\text{-obj} \rangle ::= \langle \text{identifier} \rangle \langle \text{one-subscr} \rangle |$   
 $\langle \text{identifier} \rangle \langle \text{two-subscr} \rangle$
- (71)  $\langle \text{one-subscr} \rangle ::= [\langle \text{alpha-string} \rangle] | [\langle \text{int-width} \rangle]$



- (72)  $\langle \text{two-subscr} \rangle ::= [\langle \text{alpha-string} \rangle :: \langle \text{alpha-string} \rangle] |$   
 $[\langle \text{int-width} \rangle :: \langle \text{int-width} \rangle]$
- (73)  $\langle \text{right-hand-expr} \rangle ::= := \langle \ell\text{-obj} \rangle | := \langle \text{simp-arith-expr} \rangle |$   
 $:= \langle \text{status-test} \rangle | := \langle \text{simp-bod-expr} \rangle |$   
 $:= \langle \text{cond-arith-expr} \rangle | := \langle \text{cond-bool-expr} \rangle |$   
 $:= \langle \text{simp-arith-expr} \rangle \langle \text{rel-op} \rangle$   
 $\langle \text{simp-arith-expr} \rangle$
- (74)  $\langle \text{simp-arith-expr} \rangle ::= \langle \text{prim-arith-expr} \rangle$
- (75)  $\langle \text{prim-arith-expr} \rangle ::= \langle \text{arith-primary} \rangle |$   
 $\langle \text{prim-arith-expr} \rangle \langle \text{arith-op} \rangle \langle \text{simp-}\ell\text{-obj} \rangle |$   
 $\langle \text{prim-arith-expr} \rangle \langle \text{arith-op} \rangle \langle \text{subscr-}\ell\text{-obj} \rangle |$   
 $\langle \text{prim-arith-expr} \rangle \langle \text{arith-op} \rangle \langle \text{integer} \rangle$
- (76)  $\langle \text{arith-primary} \rangle ::= \langle \ell\text{-obj} \rangle \langle \text{arith-op} \rangle \langle \text{simp-}\ell\text{-obj} \rangle |$   
 $\langle \ell\text{-obj} \rangle \langle \text{arith-op} \rangle \langle \text{subscr-}\ell\text{-obj} \rangle |$   
 $\langle \ell\text{-obj} \rangle \langle \text{arith-op} \rangle \langle \text{signed-int} \rangle |$   
 $\langle \ell\text{-obj} \rangle \langle \text{arith-op} \rangle \langle \text{integer} \rangle |$   
 $\langle \text{signed-int-width} \rangle \langle \text{arith-op} \rangle \langle \text{simp-}\ell\text{-obj} \rangle |$   
 $\langle \text{signed-int-width} \rangle \langle \text{arith-op} \rangle \langle \text{subscr-}\ell\text{-obj} \rangle |$   
 $\langle \text{int-width} \rangle \langle \text{arith-op} \rangle \langle \text{subscr-}\ell\text{-obj} \rangle |$   
 $\langle \text{int-width} \rangle \langle \text{arith-op} \rangle \langle \text{simp-}\ell\text{-obj} \rangle$
- (77)  $\langle \text{signed-int-width} \rangle ::= \langle \text{signed-int} \rangle$
- (78)  $\langle \text{signed-int} \rangle ::= \underline{\text{neg}} \langle \text{integer} \rangle$
- (79)  $\langle \text{arith-op} \rangle ::= + | - | \underline{\text{plus}} | \underline{\text{min}} | \underline{\text{add}} | \underline{\text{sub}}$





- (80)  $\langle \text{simp-bool-expr} \rangle ::= \langle \text{prim-bool-expr} \rangle$
- (81)  $\langle \text{prim-bool-expr} \rangle ::= \langle \text{bool-primary} \rangle | \langle \text{mon-expr} \rangle$   
 $\langle \text{prim-bool-expr} \rangle \langle \text{dy-bool-op} \rangle \langle \text{simp-l-obj} \rangle |$   
 $\langle \text{prim-bool-expr} \rangle \langle \text{dy-bool-op} \rangle \langle \text{subscr-l-obj} \rangle |$   
 $\langle \text{prim-bool-expr} \rangle \langle \text{dy-bool-op} \rangle \langle \text{bin-const} \rangle$
- (82)  $\langle \text{bool-primary} \rangle ::= \langle \text{l-obj} \rangle \langle \text{dy-bool-op} \rangle \langle \text{simp-l-obj} \rangle |$   
 $\langle \text{l-obj} \rangle \langle \text{dy-bool-op} \rangle \langle \text{subscr-l-obj} \rangle |$   
 $\langle \text{l-obj} \rangle \langle \text{dy-bool-op} \rangle \langle \text{bin-const} \rangle |$   
 $\langle \text{bin-int} \rangle \langle \text{dy-bool-op} \rangle \langle \text{simp-l-obj} \rangle |$   
 $\langle \text{bin-int} \rangle \langle \text{dy-bool-op} \rangle \langle \text{subscr-l-obj} \rangle |$
- (83)  $\langle \text{mon-expr} \rangle ::= (\langle \text{mon-bool-op} \rangle \langle \text{l-obj} \rangle)$
- (84)  $\langle \text{bin-int} \rangle ::= \langle \text{bin-const} \rangle$
- (85)  $\langle \text{bin-const} \rangle ::= \text{'\langle bit-string \rangle'}$
- (86)  $\langle \text{bit-string} \rangle ::= \langle \text{bin-digit} \rangle | \langle \text{bit-string} \rangle \langle \text{bin-digit} \rangle$
- (87)  $\langle \text{bin-digit} \rangle ::= 1/0$
- (88)  $\langle \text{dy-bool-op} \rangle ::= \wedge | \vee | \oplus$
- (89)  $\langle \text{mon-bool-op} \rangle ::= \neg | \underline{\text{shl}} | \underline{\text{shr}} | \underline{\text{shli}} | \underline{\text{shri}} | \underline{\text{shlc}} | \underline{\text{shrc}}$
- (90)  $\langle \text{cond-arith-expr} \rangle ::= \langle \text{if-expr} \rangle \langle \text{then-arith-expr} \rangle$   
 $\langle \text{else-arith-expr} \rangle$
- (91)  $\langle \text{if-expr} \rangle ::= \underline{\text{if}} \langle \text{relation} \rangle | \underline{\text{if}} \langle \text{status-test} \rangle$



- (92)  $\langle \text{relation} \rangle ::= \langle \text{simp-arith-expr} \rangle \langle \text{rel-op} \rangle \langle \text{simp-arith-expr} \rangle$
- (93)  $\langle \text{rel-op} \rangle ::= = | \neq | < | > | \leq | \geq$
- (94)  $\langle \text{status-test} \rangle ::= \underline{\text{test}} \text{ identifier}$
- (95)  $\langle \text{then-arith-expr} \rangle ::= \underline{\text{then}} \langle \text{simp-arith-expr} \rangle$
- (96)  $\langle \text{else-arith-expr} \rangle ::= \underline{\text{else}} \langle \text{simp-arith-expr} \rangle |$   
 $\underline{\text{else}} \langle \text{cond-arith-expr} \rangle$
- (97)  $\langle \text{cond-bool-expr} \rangle ::= \langle \text{if-expr} \rangle \langle \text{then-bool-expr} \rangle$   
 $\langle \text{else-bool-expr} \rangle$
- (98)  $\langle \text{then-bool-expr} \rangle ::= \underline{\text{then}} \langle \text{simp-bool-expr} \rangle$
- (99)  $\langle \text{else-bool-expr} \rangle ::= \underline{\text{else}} \langle \text{simp-bool-expr} \rangle |$   
 $\underline{\text{else}} \langle \text{cond-bool-expr} \rangle$
- (100)  $\langle \text{goto-st} \rangle ::= \underline{\text{goto}} \langle \text{identifier} \rangle$
- (101)  $\langle \text{cond-st} \rangle ::= \langle \text{if-then-st} \rangle | \langle \text{if-then-else-st} \rangle$
- (102)  $\langle \text{if-then-st} \rangle ::= \langle \text{if-expr} \rangle \langle \text{then-st} \rangle$
- (103)  $\langle \text{if-then-else-st} \rangle ::= \langle \text{if-expr} \rangle \langle \text{then-st} \rangle \langle \text{else-st} \rangle$
- (104)  $\langle \text{then-st} \rangle ::= \underline{\text{then}} \langle \text{assign-st} \rangle | \underline{\text{then}} \langle \text{goto-st} \rangle | \underline{\text{then}} \langle \text{block} \rangle |$   
 $\underline{\text{then}} \langle \text{file-st} \rangle | \underline{\text{then}} \langle \text{file-ctl-st} \rangle | \underline{\text{then}} \langle \text{proc-st} \rangle$
- (105)  $\langle \text{else-st} \rangle ::= \underline{\text{else}} \langle \text{assign-st} \rangle | \underline{\text{else}} \langle \text{goto-st} \rangle | \underline{\text{else}} \langle \text{block} \rangle |$   
 $\underline{\text{else}} \langle \text{file-st} \rangle | \underline{\text{else}} \langle \text{file-ctl-st} \rangle | \underline{\text{else}} \langle \text{for-st} \rangle |$   
 $\underline{\text{else}} \langle \text{if-then-else-st} \rangle | \underline{\text{else}} \langle \text{proc-st} \rangle$



- (106)  $\langle \text{for-st} \rangle ::= \underline{\text{for}} \langle \ell\text{-obj} \rangle \langle \text{from-cl} \rangle \langle \text{step-cl} \rangle$   
 $\langle \text{until-cl} \rangle \langle \text{do-cl} \rangle$
- (107)  $\langle \text{from-cl} \rangle ::= \underline{\text{from}} \langle \ell\text{-obj} \rangle | \underline{\text{from}} \langle \text{integer} \rangle$
- (108)  $\langle \text{step-cl} \rangle ::= \underline{\text{step}} \langle \ell\text{-obj} \rangle | \underline{\text{step}} \langle \text{integer} \rangle$
- (109)  $\langle \text{until-cl} \rangle ::= \underline{\text{until}} \langle \ell\text{-obj} \rangle | \underline{\text{until}} \langle \text{integer} \rangle$
- (110)  $\langle \text{do-cl} \rangle ::= \underline{\text{do}} \langle \text{assign-st} \rangle | \underline{\text{do}} \langle \text{block} \rangle$
- (111)  $\langle \text{file-st} \rangle ::= \underline{\text{get}} \langle \text{file-name} \rangle \underline{\text{to}} \langle \ell\text{-obj-list} \rangle |$   
 $\underline{\text{put}} \langle \ell\text{-obj} \rangle \underline{\text{into}} \langle \text{file-name} \rangle$
- (112)  $\langle \text{file-ctl-st} \rangle ::= \underline{\text{init}} \langle \text{file-name} \rangle | \underline{\text{init}} \langle \text{file-name} \rangle$   
 $[\langle \text{alpha-string} \rangle] | \langle \text{other-file-ctl-st} \rangle$
- (113)  $\langle \text{other-file-ctl-st} \rangle ::= \langle \text{file-ctl-op} \rangle \langle \text{file-name} \rangle$
- (114)  $\langle \text{file-ctl-op} \rangle ::= \underline{\text{term}} | \underline{\text{rwd}} | \underline{\text{bsp}}$
- (115)  $\langle \text{block} \rangle ::= \underline{\text{begin}} \langle \text{decl-list} \rangle \langle \text{st-list} \rangle \underline{\text{end}} | \underline{\text{begin}} \langle \text{st-list} \rangle \underline{\text{end}}$
- (116)  $\langle \text{decl-list} \rangle ::= \langle \text{data-obj-list} \rangle$
- (117)  $\langle \text{data-obj-list} \rangle ::= \langle \text{decl} \rangle ; | \langle \text{decl} \rangle ; \langle \text{data-obj-list} \rangle$
- (118)  $\langle \text{decl} \rangle ::= \langle \ell\text{-obj-decl} \rangle | \langle \text{time-unit-decl} \rangle | \langle \text{phase-decl} \rangle |$   
 $\langle \text{bmc-decl} \rangle | \langle \text{stc-decl} \rangle | \langle \text{phase-rel-decl} \rangle |$   
 $\langle \text{file-decl} \rangle | \langle \text{operator-decl} \rangle | \langle \text{file-op-decl} \rangle$
- (119)  $\langle \text{st-list} \rangle ::= \langle \text{exec-list} \rangle$
- (120)  $\langle \text{exec-list} \rangle ::= \langle \text{statement} \rangle | \langle \text{statement-str} \rangle \langle \text{exec-list} \rangle$



(121) <statement-str> ::= <statement>;

(122) <proc-st>:: = call<identifier>|call<identifier>  
                  (<param-list>)

(123) <param-list> ::= <ℓ-obj-list>

(124) <proc-decl> ::= proc<identifier><block>|  
                      proc<identifier>(<param-list>)<block>





## APPENDIX B

### EXAMPLES OF MICROL PROGRAMS

Some examples of microprograms using MICROL are presented in this Appendix. Two processors are considered here: Microdata Corporation's MICRO-1600 [41], and the IBM System/360 Model 30 [19]. In the microprogram examples, comments are written in the form '#S#', where 'S' is a string of characters.

```
begin # MICRO-1600 description follows. For further
reference, see [41] #
loc M = loc N = loc OD = loc FLAGR = loc GPR = loc U = loc T =
 seq(8) bit;
loc WORD = loc RH = loc RL = seq (8) bit;
loc MEM = seq (4096) WORD;
loc FILER = seq (16) GPR;
loc MAR = (M,N);
loc R = (RH,RL);
log LREG = loc LSAVE = seq (12) bit;
loc IOC = seq (3) bit;
loc AL = loc ML = bit; loc LINK = (AL,ML);
start file declaration
file TTR01 with (KBF1);
file TTR02 with (KBF2);
time-object declaration
phase P = (0:200) nanosec: phase IOP = (0:1000) nanosec;
```



```

bmc CYCLE = (P);
phase MP1 = (0:400) nanosec;
phase MP2 = (400:1000) nanosec;
rphase RD = (MP1);
wphase WTE = (MP2);
stc MC = (RD,WTE);
sync MC with CYCLE; sync IOP with CYCLE;
operator declarations
op:=width 8 bet RD and RD from (MEM) to (T);
op:=width 8 bet MC and MC from (T) to (MEM);
op:=width 8 bet CYCLE and CYCLE from (T) to (OD);
op:=width 8 bet CYCLE and CYCLE from (T,RL,FILER,FLAGR,LINK)
 to (LREG,M,N,FILER,T);
op +,- width 8 bet CYCLE and CYCLE from (T,RL,LINK):
 (FILER,FLAGR) to (LREG,M,N,FILER):(LINK);
+ and - uses two's complement arithmetic
op ^,∨,⊕ width 8 bet CYCLE and CYCLE from (T,RL):(FILER,FLAGR)
 to (LREG,M,N,FILER);
op shli,shlr,shl,shr width 8 bet CYCLE and CYCLE from (FILER)
 to (LREG,M,N,FILER):(LINK);
op ptr width 0 bet 0 and 0 from (MAR) to (MEM);
file operator declaration
fop get width 8 bet IOP and IOP from (TTR01,TTR02) to (T);
Example microprograms. "Example 1": Multiply two positive
numbers. Each number 8 bits maximum including sign. The
result is to occupy two 8-bit file registers. The numbers

```



are in file registers before the multiply is executed.  
The algorithm is add and shift #

begin

loc LOOPCTR = FILER[8]; loc MPLR = FILER[2];

loc MPLND = FILER[3];

LOOPCTR := 8;

T := MPLR;

FILER [4] := 0;

L1: if MPLND [8]  $\neq$  0 then FILER [4] := T + FILER [4];

FILER [4] := (shr FILER [4]);

MPLND := (shri MPLND); LOOPCTR := LOOPCTR - 1;

if LOOPCTR  $\neq$  0 then goto L1;

end

# "Example 2" - Data input from two input devices to  
memory locations '200' to '207'. The two devices pass  
their data alternatively. Assume the address '200' to  
be in file register 10 and 11 #

begin

loc SWITCH = FILER [15];

SWITCH := 1;

L2: M := FILER [11];

N := FILER [10];

if SWITCH = 1 then begin

get TTR01 to MEM [MAR];

FILER [10] := FILER [10] + 1;

SWITCH := 2;

end



else begin

get TTR02 to MEM [MAR];

FILER [10] := FILER [10] + 1;

SWITCH := 1;

end

if FILER [10] < 8 then goto L2;

end

# "Example 3" - 16-bit add, core-to-file. This routine adds the contents of file registers AU and AL to a 16 bit word in main memory at the address contained in registers OU and OL, and places the result in AU and AL. File designations: AU = FILER [4]; AL = FILER [5]; OU = FILER [8]; OL = FILER [9] #

begin

M := FILER [8]; N := FILER [9];

FILER [1] := MEM [MAR];

FILER [9] := FILER [9] + 1;

if LINK = 1 then begin

FILER [8] := FILER [8] + 1;

end

FILER [5] := FILER [5] + MEM [MAR];

T := FILER [1]; FILER [10] := LINK;

FILER [4] := FILER [4] + T + FILER [10];

end

# "Example 4" - Input a 32-bit word from an external device to main memory. The input device is a paper tape reader which inputs 8 bits at a time. Core memory address





is in registers OU = FILER [8], OL = FILER [9] #

begin

M := FILER [8];

for FILER [1] from 1 step 1 until 4 do

begin N := FILER [9];

get PTAPER1 to MEM [MAR];

FILER [9] := FILER [9] + 1; FILER[11] := LINK;

FILER [8] := FILER [8] + 1 + FILER [11];

end

end

# "Example 6": Conversion of 3-digit BCD code plus sign to binary. Sign is DS, the three digits are D2, D1, D0. DS and D2 are in BU = FILER [4]; D1 and D0 are in BL = FILER [5]. Binary result will be in AU = FILER [7], and AL = FILER [8]. The technique is to multiply each BCD digit by its power of 10 expressed in binary, and to add each converted digit value in an accumulator. A multiply procedure is called for multiplication. For further reference, see [41] #

begin

loc BU = FILER [4]; loc BL = FILER [5];

loc AU = FILER [7]; loc AL = FILER [8];

loc OP = FILER [1]; # contains digit value #

loc V = FILER [10]; # contains power of 10 in binary #

loc W = FILER [11]; # dummy loop ctr #



```

AL := 15; T := BL; AL := AL ^ T; AU := 0; OP := T;
OP contains lower two digits
for W from 1 step 1 until 4 do OP := (shr OP);
T := 15; OP := OP ^ T; V := 10;
call MPY (OP,T,V,AL,AU,LINK);
OP := 15; T := BU; OP := T ^ OP; V := 100;
call MPY (OP,T,V,AL,AU,LINK);
if BU [1] = 1 then goto LL4;
AL := AL \oplus '11111111'; # one's complement of AL #
AL := AL + 1; # two's complement of AL #
AU := AU \oplus '11111111';
AU := AU + LINK;
goto LL4;
proc MPY (OP,T,V,AL,AU,LINK); # the formal and actual
parameters are identical #
 begin
LL2 : if OP = 0 then goto LL3;
 T := V;
 AL := AL + T;
 AU := AU + LINK;
 OP := OP - 1;
 goto LL2;
LL3 : end
LL4 : end

```



begin

# Example MICROL program for storage-to-register-add operation in System/360 Model 30. Instruction format is "Op, Reg, Storage Address". The register is within main storage. For further reference, see Iverson and Brooks, "Automatic Data Processing", John Wiley, 1969, pp.259-260 #

# declarations - l-objects #

loc BYTE = loc M = loc N = loc U = loc V = loc R = loc T =  
loc G = loc A = loc B = seq (8) bit;

loc PCTR = (U,V);

loc MAR = (M,N);

loc MEM = seq (8192) BYTE;

loc ZBUS = union (U,V,R,T,D,G);

loc BBUS = union (R,D);

# time-object declarations #

phase P = (0:750) nanosec;

rphase RD = (P);

wphase WT = (P);

bmc MC = (P);

sync RD with MC;

sync WT with MC;

# operator declarations #

op +,- width 8 bet MC and MC from (ZBUS):(BBUS) to (ZBUS);

op := width 8 bet MC and MC from (ZBUS,BBUS) to (ZBUS);



```

op ptr width 0 bet 0 and 0 from (MAR) to (MEM);
op := width 8 bet MC and MC from (U) to (M);
op := width 8 bet MC and MC from (V,T) to (N);
op := width 8 bet RD and RD from (MEM) to (R);
op := width 8 bet WT and WT from (R) to (MEM);
it is assumed that the instruction has been fetched,
and the effective storage address calculated #
AB1 : MAR := PCTR;
 D := MEM [MAR];
 MEM [MAR] := R; # memory regeneration #
 N := T; M := 0; V := V - 1;
 R := MEM [MAR];
 MEM [MAR] := R + D; T := T - 1;
 if V \neq 0 then goto AB1;
end

```

















**B30080**